International Journal of Software Engineering and Knowledge Engineering

World Scientific
www.worldscientific.com

A SOFTWARE RELIABILITY MODEL FOR CLOUD-BASED SOFTWARE REJUVENATION USING DYNAMIC FAULT TREES

JEAN RAHME[*] and HAIPING XU[†]

Computer and Information Science Department, University of Massachusetts Dartmouth
North Dartmouth, MA 02747, USA
[*]*jrahme@umassd.edu*
[†]*hxu@umassd.edu*

Received (28 August 2015)
Revised (18 October 2015)
Accepted (Day Month Year)

Correctly measuring the reliability and availability of a cloud-based system is critical for evaluating its system performance. Due to the promised high reliability of physical facilities provided for cloud services, software faults have become one of the major factors for the failures of cloud-based systems. In this paper, we focus on the software aging phenomenon where system performance may be progressively degraded due to exhaustion of system resources, fragmentation and accumulation of errors. We use a proactive technique, called software rejuvenation, to counteract the software aging problem. The dynamic fault tree (DFT) formalism is adopted to model the system reliability before and during a software rejuvenation process in an aging cloud-based system. A novel analytical approach is presented to derive the reliability function of a cloud-based Hot SPare (HSP) gate, which is further verified using Continuous Time Markov Chains (CTMC) for its correctness. We use a case study of a cloud-based system to illustrate the validity of our approach. Based on the reliability analytical results, we show how cost-effective software rejuvenation schedules can be created to keep the system reliability consistently staying above a predefined critical level.

Keywords: Software aging; software rejuvenation; reliability analysis; dynamic fault tree (DFT); hot spare (HSP) gate; Markov chain; scheduling.

1. Introduction

Due to recent advances in cloud computing technologies, cloud services have been used in many different areas such as traffic control, real-time sensor networks, healthcare, and mobile cloud computing. Cloud service providers have tried to deliver products with high quality of services (QoS), which provide users fault-tolerant hardware and reliable software platforms for deploying cloud-based applications [1][2]. However, cloud outages are still very common due to component failures, which can affect quite negatively the revenue of cloud-based systems. Previous research on the reliability of computer-based systems has focused on hardware reliability and availability; consequently, the hardware fault tolerance and fault management are well understood and developed [3]. With the promised high reliability and availability of physical facilities,

[†] Corresponding author: Dr. Haiping Xu, Associate Professor, Computer and Information Science Department, University of Massachusetts Dartmouth, Email: hxu@umassd.edu.

including the hardware facilities and their associated redundancy mechanisms, software faults have now become one of the major factors of failures in a cloud-based system. Since software reliability is considered one of the weakest points in system reliability, software fault tolerance and failure forecasting require more attentions than hardware fault tolerance in modern computer systems [4, 5]. This work is motivated to deal with the software faults in cloud computing in order to assure high reliability and availability of cloud-based software systems.

In many safety-critical computer-based systems, failures of the software systems may lead to unrecoverable loss such as human life [6]. Such systems are required to be perfectly reliable and never fail based on the discipline of fault-tolerant and reliable computing. Reliability and availability are two common ways to express system fault tolerance in industry. A reliable computer-based system typically has high availability if unreliability is the major cause for unavailability. In this paper, we focus on analyzing the reliability of cloud-based systems for software fault tolerance in software reliability engineering (SRE). Traditional SRE has been based on analysis of software defects and bugs such as Bohrbugs or Heisenbugs without considering software aging related bugs [4]. Bohrbugs are mainly design defects that can be eliminated by debugging or adopting design diversity; while Heisenbugs are defined as faults that would stop causing failure when one attempts to isolate them. The concept of software aging phenomenon was introduced in the middle 90s, which explains that the system resources used by the software degrade gradually as a function of time [7]. Software aging starts to show up due to multiple factors such as memory bloating, memory leaks, unterminated threads, data corruption, unreleased file-locks, fragmentation in storage space, and accumulation of round-off errors when running a piece of software. It has considerably changed the SRE field of study, and becomes a major factor for the reliability of fully tested and deployed software systems. To deal with the software aging problem and to assure software fault tolerance, software rejuvenation process has been introduced as a proactive approach to counteracting software aging and maintaining a reliable software system [8]. Software rejuvenation involves actions such as stopping the running software occasionally, and cleaning its internal state (e.g., garbage collection, flushing operating system kernel tables, and reinitializing internal data structures). The simplest way to perform software rejuvenation is to restart the software component that causes the aging problem, or to reboot the whole system.

Due to the ever-growing cloud computing technology and its vast markets, the workload of cloud-based systems has increased dramatically. A heavy workload of a cloud-based system will inevitably lead to more software aging problems. In this paper, we propose to use cloud-based spare components as major software components in a computer-based system to enhance its system reliability, and introduce an analytical-based approach to developing rejuvenation schedules for cloud-based systems in order to maintain their high system reliability and ensure a zero-downtime rejuvenation process. Dynamic Fault Trees (DFT) are adopted to model the reliability of a cloud-based system, and a novel analytical approach is presented to derive the reliability function of a major

type of dynamic gate in DFT models, called Hot SPare (HSP) gate. The analytical approach is then formally verified using a Continuous Time Markov Chains (CTMC) model to ensure its correctness. As the CTMC approach has its intrinsic limitation of only supporting components with constant failure rates, to the best of our knowledge, our proposed analytical approach is the first formal way to correctly derive the reliability function of an HSP gate without such a limitation. To demonstrate the practical usage of our approach in evaluating the system reliability of a cloud-based system, we assume a reliability threshold for the system under consideration. When the threshold is reached, the software rejuvenation process is triggered, and the reliability of the cloud-based system is boosted to its initial state. Our case study shows that software rejuvenation scheduling based on the reliability analysis of a cloud-based system can significantly enhance its system reliability and availability.

This work extends our previously proposed approach to producing a reliability-based software rejuvenation schedule for cloud-based systems [9]. In our previous work, we use CTMC to derive the reliability function of an HSP gate for cloud-based systems. To overcome the limitation of the CTMC approach, in this paper, we present a new analytical approach, which is more general and intuitive, and may potentially support software components with non-constant failure rates in our future research.

The rest of the paper is organized as follows. Section 2 discusses previous work related to our research. Section 3 presents a motivating example for rejuvenation of cloud-based components. Section 4 describes how to model and analyze the reliability of cloud-based systems using DFT. Section 5 presents a case study to demonstrate the validity of our approach, and Section 6 concludes the paper and mentions future work.

2. Related Work

In 1995, researchers introduced the so-called software rejuvenation technique to deal with aging-related software faults [8]. This technique, in contrast to reactive approaches with actions taken only after a software failure, is considered a proactive approach that preemptively restarts the aging application and clean software aging related bugs [10, 11]. Previous studies on software aging and software rejuvenation for predicting a rejuvenation schedule can be classified into two categories, namely analytical-based and measurement-based approaches [10]. In an analytical-based approach, a failure distribution is assumed for software faults related to the software aging phenomenon, and software rejuvenation is executed at a fixed interval based on the analytical results of the system reliability and availability. Several analytic models have been proposed to determine the optimal time for rejuvenation. Bobbio *et al.* proposed a fine-grained software degradation model for optimal rejuvenation policies [12]. Based on the assumption that the current degradation level of the system can be identified, they presented two different strategies to determine whether and when to rejuvenate. Vaidyanathan *et al.* presented an analytical model of a software system using inspection-based software rejuvenation [13]. In their proposed approach, they showed that inspection-based maintenance was advantageous in many cases over non-inspection

based maintenance. Dohi *et al.* introduced a modified stochastic model to estimate the software rejuvenation schedule [14]. The proposed model is based on semi-Markov processes, which can maximize the system availability. Koutras and Platis applied the software rejuvenation technique to cluster systems in order to achieve their high availability [15]. In their approach, software rejuvenation is carried out when a software deployed on a node starts to experience degradation; thus an unscheduled reboot may be avoided. Although the above approaches introduce various models for software rejuvenation, they are not intended to address complex system components' behaviors and interactions, such as dynamic relationships between software components including sparing relationship and functional dependency. Different from the existing analytical-based approaches, we focus on the dynamic behaviors of software components in the context of cloud-based systems. We adopt the sparing relationship as an example to demonstrate how dynamic relationships of software components in a cloud-based system can be modeled and analyzed using DFT.

On the other hand, measurement-based approach applies statistical analysis to the measured data of resource usage and degradation that may lead to the software aging problem. In a measurement-based approach, a monitoring program is used to continuously collect the system performance data, and analyze them in order to estimate the system degradation level. When exhaustion reaches a critical level, the software rejuvenation process is triggered. Machida *et al.* used Mann-Kendall test to detect software aging from traces of computer system metrics [16]. They tested for existence of monotonic trends in time series, which are often considered indication of software aging. Grottke *et al.* studied the resource usage in a web server subject to an artificial workload [17]. They applied non-parametric statistical methods to detect and estimate trends in the data sets for predicting future resource usage and software aging issues. Guo *et al.* proposed a software aging trend prediction method based on user intention [18]. The approach can be used to predict the trend of software aging based on the quantity of user requests to software components while the system is functioning. The existing measurement-based approaches are feasible ways to detect software aging problems in real-world computer-based systems, but they typically require to process large amounts of system data. Thus, they are not as efficient as analytical-based approaches. However, measurement-based approaches do provide useful insights about dynamic system behaviors and failure distributions related to software aging. As such, our research is complementary to the existing research efforts on measurement-based software rejuvenation technique that investigates the relationship of software metrics and software aging related software faults using statistical analysis [19].

Other related work attempted to address the software aging issues in virtualized datacenters. Machida *et al.* proposed a Petri net based availability model for virtualized systems with time-based rejuvenation for virtual machines [20]. They compared three techniques in terms of steady-state availability, and suggested the optimal combination of rejuvenation trigger intervals for each rejuvenation technique using a gradient search method. Thein *et al.* proposed an analytical approach that models availability for

application servers [21]. Based on the availability model, they presented possible combinations of virtualization, high availability cluster and software rejuvenation. However, the above approaches are not explicitly based on software reliability analysis. In contrast, our approach analyzes system reliability using DFT models, and can generate rejuvenation schedules that explicitly satisfy the predefined reliability and availability requirements of a cloud-based system.

3. Rejuvenation of Cloud-Based Components

Virtualization technology has been well-adopted in cloud computing, which allows one to share a machine's physical resources among multiple virtual environments, called virtual machines (VM). As shown in Fig. 1, A VM is not bounded to the hardware directly; rather it is bounded to generic drivers that are created by a virtual machine manager (VMM) or a hypervisor. Since a VM can be easily created and destroyed, it is particularly useful in a disaster recovery process of a cloud-based system. In this paper, a cloud-based system is referred to as a software system that consists of multiple VMs, where each VM is considered a software component within the system.

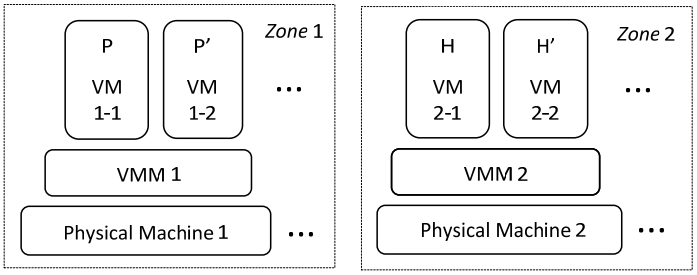

Fig. 1. An example of a reliable cloud-based system with spare software components

As a proactive fault management technique, software rejuvenation has been used to refresh system internal states and prevent the occurrence of software failures due to software aging. As mentioned before, a simple way for software rejuvenation is system reboot, e.g., to restart a VM or all VMs in a cloud-based system. The basic idea of our approach is to create a new instance of VM that replaces the one to be rejuvenated. Since the newly deployed VM instance has not yet been affected by the software aging phenomenon, the reliability of the software component, after being replaced, is boosted back to its initial condition. To achieve high fault tolerance and reliability, we further adopt the software redundancy technique using two different types of software standby spares, namely Cold SPare (CSP) and HSP. In the context of cloud computing, cold standby means that a software component is available as an image of a VM, rather than an active VM instance. Data between a primary component and the spare one is regularly mirrored based on a specified schedule, e.g., multiple times a day. Since a CSP is not up running and does not take any workload, its reliability equals to 1 with a constant failure rate 0. Since a CSP can be started very quickly, the recovery time using CSP typically

takes just a few minutes to no more than two hours. Note that a software-defined CSP is quite difference from a hardware-based CSP in terms of its cost and efficiency. The cost of a software-defined CSP is its storage and very little CPU time for data mirroring; while a hardware-based CSP is a physical device that must be available all the time in order to assure fast failover [3]. Furthermore, a software-defined CSP can be started very quickly, but a hardware-based CSP typically requires manual configuration and adjustment in the event of partial or total failure.

On the other hand, an HSP in the context of cloud computing is a hot standby VM instance. This means that the software component serving as an HSP must be installed and deployed, and must be instantly available when the primary component fails. Although an HSP is deployed and running along with the primary component, it typically does not take any workload for processing user requests. To ensure fault tolerance, critical data of an HSP is mirrored in near real time (e.g., in the range of 200 μs) from the primary VM instance. This generally provides a recovery time of a few seconds in case of a failure. Similar to CSP, a software-defined HSP also has much lower cost and works more efficiently than a hardware-based HSP. In our system design, each critical primary component must be equipped with at least one HSP and one CSP in order to maintain the needed reliability. However, when calculating the system reliability, we only need to consider the primary component and its HSP, but not its CSP, as the CSP is not functioning. A CSP is considered for reliability analysis only when it becomes a primary component or a hot spare one. In the following, for simplicity, we denote a primary VM instance/component as *P*, which is active and has a full workload, an HSP as *H*, which is active but does not take any workload, and a CSP as *C*, which is inactive and not functioning at all.

In our approach, a rejuvenation schedule of a cloud-based system is created based on its reliability modeling and the analytical results. When the reliability of a system component or the whole system reaches a predefined threshold, the rejuvenation process is triggered. We assume the rejuvenation process takes about 30 minutes, which is typically sufficient for starting a CSP and transfer all requests to the new VM. As a simple example illustrated in Fig. 1, suppose we have two instances, a primary component *P* and a hot standby one *H*, which are deployed on two different physical machines. The two physical machines usually belong to two different zones (denoted as Zone 1 and Zone 2 in Fig. 1), so a power/network outage in one zone will not affect the availability of the other one [22]. To rejuvenate the whole system, we can start two CSPs *C*1 and *C*2, denoted as *P'* and *H'* in Fig. 1, to replace *P* and *H*, respectively. Although in Fig. 1, *P'* and *H'* are deployed on the same physical machine where *P* and *H* are deployed, respectively, in reality, this is not necessary and both *P'* and *H'* can be deployed on any physical servers.

Once the spare components *P'* and *H'* are up and running, *P'* serves as a new primary component and starts to process new user requests; while *H'* serves as a new HSP, which is kept alive but does not take any workload. Meanwhile, we allow 30 minutes in total for the old components *P* and *H* to finish processing their existing requests. After 30 minutes,

we shut down and delete the components *P* and *H*, which shall have been successfully replaced by *P'* and *H'* after the rejuvenation process completes. Finally, two new CSPs *C*1 and *C*2 are created and made ready for the next round of a rejuvenation process. Note that in our rejuvenation strategy, we have chosen to shut down instances *P* and *H* rather than restart and reuse them. This is because different from a physical machine, a VM can be easily created and deployed, thus deploying new instances *P'* and *H'* is a much more efficient way than restarting and reusing *P* and *H*.

During the rejuvenation procedure, we need to consider two scenarios. One scenario is to rejuvenate the major software components all together. In this case, we replicate the whole system when the system reliability reaches its threshold. We call this scenario a system-specific rejuvenation. The second scenario is a component-specific one, where each time we only rejuvenate the critical component whose reliability is typically the lowest one when the system reliability reaches its reliability threshold. As we can see from a case study presented in Section 5, the component-specific rejuvenation would be normally more cost-effective than the system-specific approach.

4. Modeling and Analysis Using DFT

In this section, we first briefly introduce DFT, and then we show how to use DFT to model and analyze the reliability of a cloud-based system subject to software rejuvenation. To simplify matters, we assume that the time-to-failure for each software component (i.e., a VM) has a probability density function (*pdf*) that is exponentially distributed; in other words, all VMs have constant failure rates.

4.1. HSP Gate for Cloud-Based Systems

The fault tree modeling technique was introduced in 1962 at Bell Telephone Lab, which provides a conceptual modeling approach to representing system level reliability in terms of interactions between component reliabilities [3]. Fault tree analysis (FTA) is by far the most commonly used technique for risk and reliability analysis, where the system failure is described in terms of the failure of its components. Standard fault trees are combinatorial models and are built using static gates (e.g., AND-gate, OR-gate, and K/M-gate) and basic events. As combinatorial models can only capture the combination of events without considering the order of occurrences of their failures, they are usually inadequate to model today's complex dynamic systems [23, 24].

DFT augments the standard combinatorial gates of a regular fault tree, and introduces three novel modeling capabilities, namely spare component management and allocation, functional dependency, and failure sequence dependency [25]. These modeling capabilities are realized using three main dynamic gates: the spare gate, the functional dependency gate, and the priority-AND gate. The work done in this paper uses the dynamic spare gates, in particular the HSP and CSP gates. Note that a spare gate has one primary input and one or more alternate inputs (i.e., the spares). The primary input is initially powered on, and when it fails, it is replaced by an alternate input. The spare gate fails when the primary and all the alternate inputs fail. Figure 2 shows an HSP gate with

one primary component denoted as P and one hot spare component denoted as H. The HSP gate fails when both of the two components P and H fail.

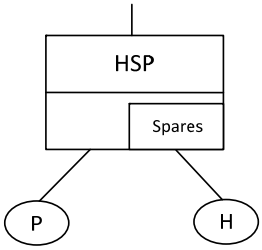

Fig. 2. A HSP gate with one primary component P and one hot spare component H

Suppose the constant failure rates of components P and H are λ_P and λ_H, respectively. Since H does not take any workload when P is functioning, its failure rate λ_H is typically lower than λ_P. When P fails, H takes over P's workload, and behaves as a primary component. H now has a higher constant failure rate λ_{H^*} than λ_H due to the software aging phenomenon with H's full workload. For this reason, we call the spare component H, after its role transition, H^*. Note that λ_{H^*} and λ_P do not have to be equal because P and H may have different configurations.

There are two scenarios when the HSP gate fails. In the first scenario, P fails before H fails. This case is illustrated as "Case 1" in Fig. 3, where P fails at τ_1 and H^* fails at τ_2, with $\tau_1 < \tau_2$. In the second scenario, H fails before P fails. In this case, H does not have a chance to behave as a primary component, and the failure of P immediately leads to the failure of the HSP gate. This case is illustrated as "Case 2" in Fig. 3, where $\tau_2 < \tau_1$.

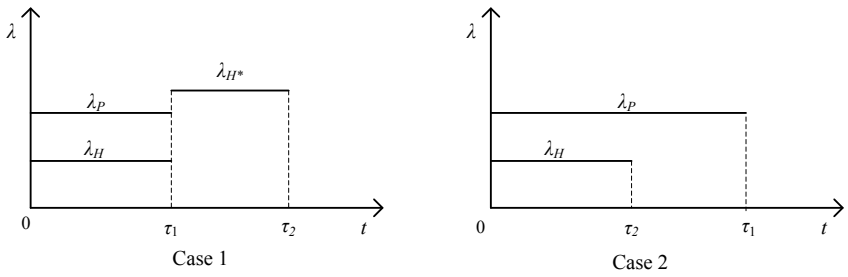

Fig. 3. Two cases for the failure of an HSP gate (Case 1: P fails before H; Case 2: H fails before P)

We now derive the reliability function $R(t)$ of the HSP gate by considering the above two cases.

Case 1: P fails before H fails, denoted as $P \prec H$. In this case, it is guaranteed that H does not fail during $(0, \tau_1]$. After P fails, H takes over the workload and becomes H^*. Intuitively, the distribution function $F_{P \prec H}(t)$ of the HSP gate, i.e., the probability that the HSP gate fails during $(0, t]$ can be calculated as in Eq. (1).

$$F_{P\prec H}(t) = \Pr_{P\prec H}(T \le t) = \int_0^t \int_{\tau_1}^t (\lambda_P e^{-\lambda_P \tau_1})(\lambda_{H*} e^{-\lambda_{H*}\tau_2}) d\tau_2 d\tau_1 \quad (1)$$

However, Eq. (1) works only when $\lambda_{H*} = \lambda_H$, i.e., the constant failure rate of H does not change after it switches its role from a spare component to a primary one at time τ_1. When $\lambda_{H*} > \lambda_H$, as we can see from Fig. 4, the integration of the *pdf* of H* from τ_1 to t does not give the correct unreliability of the component at time t, because it incorrectly assumes that the component behaves as H* starting from time 0. Since the component actually behaves as H during (0, τ_1], the unreliability of H* at time τ_1 equals the unreliability of H at τ_1 rather than the unreliability calculated by the integration of the *pdf* of H* from 0 to τ_1. This requires us to calculate a new starting integration time τ_{H*} for H* such that the unreliability of H* at τ_{H*} (represented by the shaded area under the *pdf* of H*) is equal to the unreliability of H at τ_1 (represented by the shaded area under the *pdf* of H). As the *pdf*s of H and H* are $f(\tau) = \lambda_H e^{-\lambda_H \tau}$ and $f(\tau) = \lambda_{H*} e^{-\lambda_{H*}\tau}$, respectively, such a relationship between H and H* can be described as in Eq. (2).

$$\int_0^{\tau_{H*}} \lambda_{H*} e^{-\lambda_{H*}\tau} d\tau = \int_0^{\tau_1} \lambda_H e^{-\lambda_H \tau} d\tau \quad (2)$$

Solving Eq. (2), we have $\tau_{H*} = \frac{\lambda_H}{\lambda_{H*}} \tau_1$. Since H* fails during a period of time (t-τ_1), the integration range for H* now becomes $[\frac{\lambda_H}{\lambda_{H*}} \tau_1,\ t - \tau_1 + \frac{\lambda_H}{\lambda_{H*}} \tau_1]$. Based on the above analysis, the probability of P fails before H fails can be calculated as in Eq. (3).

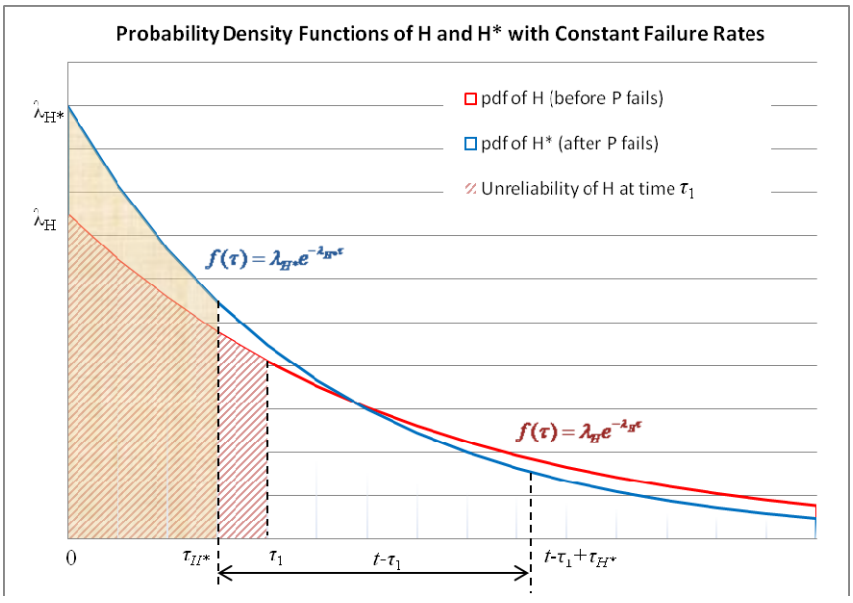

Fig. 4. The initial unreliability of H* when P fails (i.e., the unreliability of H at time τ_1)

$$F_{P\prec H}(t) = \Pr_{P\prec H}(T \le t) = \int_0^t \int_{\frac{\lambda_H}{\lambda_{H^*}}\tau_1}^{t-\tau_1+\frac{\lambda_H}{\lambda_{H^*}}\tau_1} (\lambda_P e^{-\lambda_P \tau_1})(\lambda_{H^*} e^{-\lambda_{H^*}\tau_2}) d\tau_2 d\tau_1 \tag{3}$$

To simplify the integration range for H^*, we can substitute $u(\tau_2) = \tau_2 - \frac{\lambda_H}{\lambda_{H^*}}\tau_1$ for variable τ_2 in Eq. (3), and derive the distribution function $F_{P\prec H}(t)$ of the HSP gate appearing in Case 1 as in Eq. (4).

$$\begin{aligned}
F_{P\prec H}(t) &= \Pr_{P\prec H}(T \le t) \\
&= \int_0^t \int_0^{t-\tau_1} (\lambda_P e^{-\lambda_P \tau_1})(\lambda_{H^*} e^{-\lambda_{H^*}(u+\frac{\lambda_H}{\lambda_{H^*}}\tau_1)}) du d\tau_1 \\
&= \int_0^t \int_0^{t-\tau_1} (\lambda_P e^{-(\lambda_P+\lambda_H)\tau_1})(\lambda_{H^*} e^{-\lambda_{H^*}u}) du d\tau_1 \\
&= \int_0^t (\lambda_P e^{-(\lambda_P+\lambda_H)\tau_1})(1 - e^{-\lambda_{H^*}(t-\tau_1)}) d\tau_1 \\
&= \frac{\lambda_P}{\lambda_P+\lambda_H}(1 - e^{-(\lambda_P+\lambda_H)t}) - \frac{\lambda_P}{\lambda_P+\lambda_H-\lambda_{H^*}} e^{-\lambda_{H^*}t}(1 - e^{-(\lambda_P+\lambda_H-\lambda_{H^*})t}) \\
&= \frac{\lambda_P}{\lambda_P+\lambda_H}(1 - e^{-(\lambda_P+\lambda_H)t}) + \frac{\lambda_P}{\lambda_P+\lambda_H-\lambda_{H^*}}(e^{-(\lambda_P+\lambda_H)t} - e^{-\lambda_{H^*}t})
\end{aligned} \tag{4}$$

Case 2: H fails before P fails, denoted as $H \prec P$. In this case, it is guaranteed that P does not fail during (0, τ_2]. The distribution function $F_{H\prec P}(t)$ of the HSP gate, i.e., the probability that the HSP gate fails during (0, t] can be calculated as in Eq. (5).

$$\begin{aligned}
F_{P\prec H}(t) &= \Pr_{H\prec P}(T \le t) \\
&= \int_0^t \int_{\tau_2}^t (\lambda_P e^{-\lambda_P \tau_1})(\lambda_H e^{-\lambda_H \tau_2}) d\tau_1 d\tau_2 \\
&= \int_0^t (e^{-\lambda_P \tau_2} - e^{-\lambda_P t})(\lambda_H e^{-\lambda_H \tau_2}) d\tau_2 \\
&= \int_0^t \lambda_H (e^{-(\lambda_P+\lambda_H)\tau_2} - e^{-\lambda_P t} e^{-\lambda_H \tau_2}) d\tau_2 \\
&= \frac{\lambda_H}{\lambda_P+\lambda_H}(1 - e^{-(\lambda_P+\lambda_H)t}) - e^{-\lambda_P t}(1 - e^{-\lambda_H t}) \\
&= 1 - e^{-\lambda_P t} - \frac{\lambda_P}{\lambda_P+\lambda_H}(1 - e^{-(\lambda_P+\lambda_H)t})
\end{aligned} \tag{5}$$

As the two cases are completely independent, the unreliability of the HSP gate at time t is the summation of the unreliability values of the two cases at time t. Thus, we derive the unreliability function $U(t)$ of the HSP gate as in Eq. (6).

$$\begin{aligned} U(t) &= F_{P\prec H}(t) + F_{H\prec P}(t) \\ &= \frac{\lambda_P}{\lambda_P+\lambda_H}(1-e^{-(\lambda_P+\lambda_H)t}) + \frac{\lambda_P}{\lambda_P+\lambda_H-\lambda_{H^*}}(e^{-(\lambda_P+\lambda_H)t} - e^{-\lambda_{H^*}t}) + \\ &\quad 1-e^{-\lambda_P t} - \frac{\lambda_P}{\lambda_P+\lambda_H}(1-e^{-(\lambda_P+\lambda_H)t}) \\ &= 1-e^{-\lambda_P t} + \frac{\lambda_P}{\lambda_P+\lambda_H-\lambda_{H^*}}(e^{-(\lambda_P+\lambda_H)t} - e^{-\lambda_{H^*}t}) \end{aligned} \tag{6}$$

Accordingly, the reliability function $R(t)$ of the HSP gate can be derived as in Eq. (7).

$$R(t) = 1 - U(t) = e^{-\lambda_P t} + \frac{\lambda_P}{\lambda_P+\lambda_H-\lambda_{H^*}}(e^{-\lambda_{H^*}t} - e^{-(\lambda_P+\lambda_H)t}) \tag{7}$$

It is worth noting that there is an obvious but subtle third case, where components P and H fail exactly at the same time, denoted as $P \equiv H$. As the probability of failure associated with the event $[T = \tau]$ is 0, i.e., the probability that either P or H fails during $[\tau, \tau]$ is 0, the unreliability of the HSP gate in the third case $P \equiv H$ must equal 0. This result can be easily derived as in Eq. (8), where P fails at time τ_1 during $(0, t]$, and H fails exactly at the same time τ_1 when P fails.

$$\Pr_{P\equiv H}(T \le t) = \int_0^t \int_{\tau_1}^{\tau_1} (\lambda_H e^{-\lambda_H \tau_2})(\lambda_P e^{-\lambda_P \tau_1}) d\tau_2 d\tau_1 = \int_0^t (e^{-\lambda_H \tau_1} - e^{-\lambda_H \tau_1})(\lambda_P e^{-\lambda_P \tau_1}) d\tau_1 = 0 \tag{8}$$

4.2. Verifying the Reliability Function Using CTMC

To formally verify the correctness of the reliability function $R(t)$ of the HSP gate derived in Section 4.1, we now use a CTMC model and solve its state equations.

Figure 5 shows the CTMC model corresponding to the HSP gate given in Fig. 2. There are four states 1 to 4 defined in the CTMC model, which are denoted as *PH*, *P*, *H**, and *FAILURE*, respectively. The state *PH* (State 1) refers to the one in which both the primary component and the hot spare one are functioning. When the hot spare component or the primary one fails, the model enters its *P* state (State 2) or *H** state (State 3), respectively. Note that we denote State 3 as *H** instead of *H* because in State 3, the hot spare component has a different failure rate as the one in State 1.

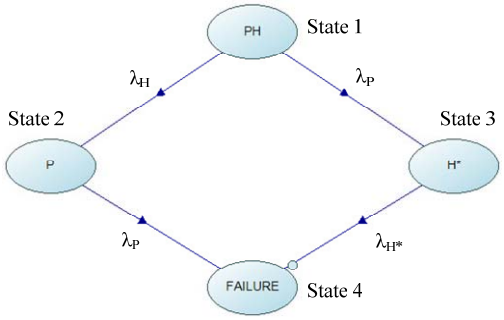

Fig. 5. The CTMC model of the HSP gate in Fig. 2

Let $P_i(t)$ be the probability of the system in state i at time t, where $1 \le i \le 4$, and $P_{ij}(dt) = P[X(t+dt) = j \mid X(t) = i]$ be the incremental transition probability with random variable $X(t)$. The following matrix $[P_{ij}(dt)]$, where $1 \le i, j \le 4$, is the incremental one-step transition matrix [4] of the CTMC defined in Fig. 5.

$$[P_{ij}(dt)] = \begin{bmatrix} 1-(\lambda_P+\lambda_H)dt & \lambda_H dt & \lambda_P dt & 0 \\ 0 & 1-\lambda_P dt & 0 & \lambda_P dt \\ 0 & 0 & 1-\lambda_{H^*} dt & \lambda_{H^*} dt \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

The matrix $[P_{ij}(dt)]$, where $1 \le i, j \le 4$, is a stochastic matrix with each row sums to 1. This matrix provides the probabilities for each state either remaining (when $i = j$) or transit to a different state (when $i \ne j$) during the time interval dt. Given the initial probabilities of the states, the matrix can be used to describe the state transition process completely. From the matrix defined in Eq. (9), we can derive the following relations as in Eqs. (10.1-10.4).

$$P_1(t+dt) = (1-(\lambda_p+\lambda_H)dt)P_1(t) \tag{10.1}$$

$$P_2(t+dt) = (\lambda_H dt)P_1(t) + (1-\lambda_p dt)P_2(t) \tag{10.2}$$

$$P_3(t+dt) = (\lambda_P dt)P_1(t) + (1-\lambda_{H^*} dt)P_3(t) \tag{10.3}$$

$$P_4(t+dt) = (\lambda_P dt)P_2(t) + (\lambda_{H^*} dt)P_3(t) + P_4(t) \tag{10.4}$$

where the initial probabilities are defined by the probability of the system being at State 1. Thus we have $P_1(0) = 1$, and $P_2(0) = P_3(0) = P_4(0) = 0$. As dt goes to 0, we derive a set of linear first-order differential equations as in Eqs. (11.1-11.4), which are state equations of the CTMC model.

$$P_1'(t) = \lim_{dt\to 0} \frac{P_1(t+dt) - P_1(t)}{dt} = -(\lambda_P+\lambda_H)P_1(t) \tag{11.1}$$

$$P_2'(t) = \lim_{dt\to 0} \frac{P_2(t+dt) - P_2(t)}{dt} = \lambda_H P_1(t) - \lambda_P P_2(t) \tag{11.2}$$

$$P_3'(t) = \lim_{dt\to 0} \frac{P_3(t+dt) - P_3(t)}{dt} = \lambda_P P_1(t) - \lambda_{H^*} P_3(t) \tag{11.3}$$

$$P_4'(t) = \lim_{dt\to 0} \frac{P_4(t+dt) - P_4(t)}{dt} = \lambda_P P_2(t) + \lambda_{H^*} P_3(t) \tag{11.4}$$

The state equations defined in Eqs. (11.1-11.4) can be solved using Laplace transformation, which allows to transform a linear first order differential equation into a linear algebraic equation that is easy to solve.

Let the Laplace transformation of $P_i(t)$ be $F_i(s)$ as defined in Eq. (12.1), the Laplace transformation of $P_i'(t)$ can be derived as in Eq. (12.2).

$$L\{P_i(t)\}(s) = \int_0^\infty e^{-st} P_i(t)dt = F_i(s) \tag{12.1}$$

$$L\{P_i'(t)\}(s) = \int_0^\infty e^{-st} P_i'(t)dt = sF_i(s) - P_i(0) \tag{12.2}$$

Now apply the Laplace transformations defined in Eqs. (12.1-12.2) to both sides of the Eqs. (11.1-11.4), we can derive Eqs. (13.1-13.4).

$$sF_1(s) - P_1(0) = -(\lambda_P + \lambda_H)F_1(s) \tag{13.1}$$
$$sF_2(s) - P_2(0) = \lambda_H F_1(s) - \lambda_P F_2(s) \tag{13.2}$$
$$sF_3(s) - P_3(0) = \lambda_P F_1(s) - (\lambda_{H^*})F_3(s) \tag{13.3}$$
$$sF_4(s) - P_4(0) = \lambda_P F_2(s) + (\lambda_{H^*})F_3(s) \tag{13.4}$$

Substituting the initial probabilities $P_i(0)$, where $1 \le i \le 4$, into Eqs. (13.1-13.4), we can solve $F_1(s)$, $F_2(s)$ and $F_3(s)$. By further applying *inverse* Laplace transformation to $F_1(s)$, $F_2(s)$ and $F_3(s)$, we can solve the original linear first order differential equations in Eqs. (10.1-10.3) as follows.

$$F_1(s) = \frac{1}{(s+\lambda_P+\lambda_H)} \Rightarrow P_1(t) = e^{-(\lambda_P+\lambda_H)t}$$
$$F_2(s) = \frac{\lambda_H}{(s+\lambda_P+\lambda_H)(s+\lambda_P)} \Rightarrow P_2(t) = e^{-\lambda_P t} - e^{-(\lambda_P+\lambda_H)t}$$
$$F_3(s) = \frac{\lambda_P}{(s+\lambda_P+\lambda_H)(s+\lambda_{H^*})} \Rightarrow P_3(t) = \frac{\lambda_P}{\lambda_P+\lambda_H-\lambda_{H^*}}(e^{-\lambda_{H^*}t} - e^{-(\lambda_P+\lambda_H)t})$$

The reliability function $R(t)$ is the summation of $P_1(t)$, $P_2(t)$ and $P_3(t)$, which can be calculated as in Eq. (14),

$$R(t) = P_1(t) + P_2(t) + P_3(t) = e^{-\lambda_P t} + \frac{\lambda_P}{\lambda_P+\lambda_H-\lambda_{H^*}}(e^{-\lambda_{H^*}t} - e^{-(\lambda_P+\lambda_H)t}) \tag{14}$$

It is easy to see that Eq. (14) gives exactly the same formula as the one defined in Eq. (7); thus, it verifies the correctness of our proposed analytical approach for calculating the reliability of the HSP gate at time t. Note that $P_4(t)$ is the probability that the system is in its *FAILURE* state at time t. Therefore, $P_4(t)$ actually defines the system unreliability function $U(t) = P_4(t) = 1 - R(t)$.

4.3. Modeling and Analysis Using DFT in Two Phases

To model and analyze the reliability of a cloud-based system with spare components, we consider two different phases. **Phase 1** represents the pre-rejuvenation stage where the reliability analysis is based on the failure rates of the primary components and their HSPs. CSPs are not considered in this phase because they cannot take over the system load instantly when both the primary and hot spare components fail. We model the

system reliability using DFT, and then calculate its reliability based on the reliability function of HSP gate derived in Section 4.1.

Phase 2 is the software rejuvenation phase. When the predefined reliability threshold is reached, the software rejuvenation process is initiated, and the system enters this phase. As we have mentioned, there are two rejuvenation scenarios, namely the system-specific rejuvenation and the component-specific one. To illustrate the basic idea of calculating the system reliability in this phase, we use the first scenario as an example, where the whole system is rejuvenated. In this scenario, we start two CSPs *P'* and *H'* to replace *P* and *H*, respectively. During the rejuvenation period, all four software components *P*, *H*, *P'* and *H'* coexist and are functioning. As shown in Fig. 6, the dynamic fault tree model is decomposed into 2 subtrees, *S*1 and *S*2, which are all HSP gates that are connected by an AND-gate. This is because the system fails only when both of the two HSP gates fail, and the failure of a single HSP gate during the rejuvenation phase will not lead to the failure of the whole system. Subtree *S*1 consists of components *P* and *H* that are to be rejuvenated; while subtree *S*2 consists of the newly deployed components *P'* and *H'*, which are used to replace *P* and *H*. As both *S*1 and *S*2 are defined as HSP gates, they can be computed using the same analysis technique as described in Phase 1.

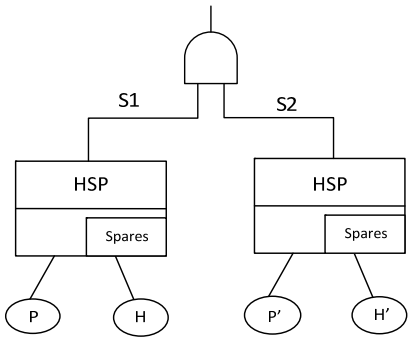

Fig. 6. A DFT model with 2 HSP gates (Phase 2)

Once we have the distribution functions of *S*1 and *S*2, the static gate, i.e., the AND-gate, can be easily solved using the sum-of-disjoint-products (SDP) method [3]. Specifically, to calculate the reliability of the whole system in this phase, we first calculate the unreliability functions $U_{S1}(t)$ and $U_{S2}(t)$ for *S*1 and *S*2, respectively. Then the reliability of the AND-gate can be calculated as in Eq. (15).

$$R(t) = 1 - U_{AND}(t) = 1 - U_{S1}(t) * U_{S2}(t) \tag{15}$$

In the following case study, we will consider both of the two scenarios during the rejuvenation process, where Scenario 1 involves rejuvenation of the whole system, and in this case, we need to replicate all major software components when the system reliability reaches the threshold. On the other hand, Scenario 2 is component specific, thus we only rejuvenate the most critical component whose reliability is the lowest when the system reliability reaches its threshold.

5. Case Study

A challenging task in cloud computing is to correctly measure the reliability of a cloud-based system and maintain its high reliability. In this case study, we show how to model and analyze the reliability of a cloud-based system using DFT, and then estimate an effective rejuvenation schedule that meets the high reliability requirement of the system. We consider a typical cloud-based system as shown in Fig. 7, which consists of an application server *PA* and a database server *PB*. To enhance the system reliability, two hot spare components *HA* and *HB* are set up for *PA* and *PB*, respectively, which are ready to take over the workload once the primary ones fail. Note that each of the servers is deployed in different zones for fault-tolerance purpose [22]. As a clarification for the reliability analysis in this case study, we view a VM with its OS, the server software and the deployed services as a single software component. In addition, we only consider the reliability of the servers within the box drawn with dashed lines, and assume the proxy server's reliability is ideal. Furthermore, we assume that the proxy server and the application server can monitor and detect failures of the application server and the database server, respectively.

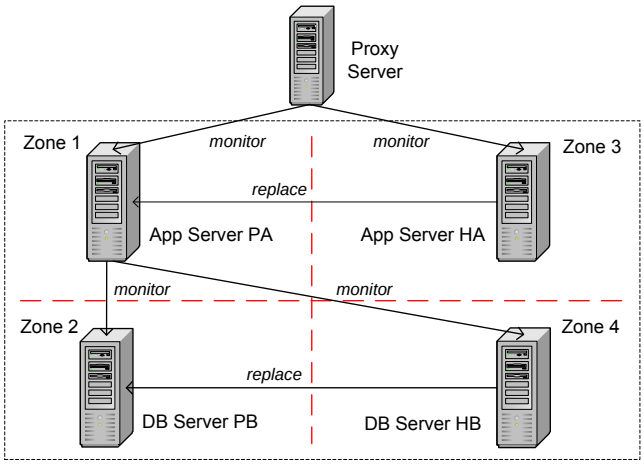

Fig. 7. A cloud-based system with 2 servers and their HSPs

To ensure a high reliability of the system, we set a reliability threshold of 0.99, and assume the constant failure rates of the servers be $\lambda_{PA} = 0.004$, $\lambda_{HA} = 0.0025$, $\lambda_{PB} = 0.005$, and $\lambda_{HB} = 0.003$. Note that the failure rates of the hot spare components are lower than their corresponding primary ones because the spare components do not take any workload when the primary ones are functioning. However, when a primary server fails, the associated hot spare component takes over the workload; in this case, its failure rate will increase accordingly. We assume the hot spare components have the same configurations as their associated primary ones, thus we have $\lambda_{HA*} = \lambda_{PA} = 0.004$ and $\lambda_{HB*} = \lambda_{PB} = 0.005$.

This case study involves 8 software components that are split into two groups. The first group consists of the four servers shown in Fig. 7. The second group consists of four CSP components that are used to replace the servers in the first group during the rejuvenation process. We name the severs in the second group as *PA'*, *HA'*, *PB'*, and *HB'*. As the CSP components are undeployed VM images, their failure rates are 0. Once deployed, they will have the same failure rates as their corresponding software components due to the assumed same configurations.

Figure 8 shows the DFT model of the cloud-based system in Phase 1. Because the system fails when either the application server or the database server fails, the two HSP gates are connected by an OR-gate. The reliability function of the OR-gate can be derived as in Eq. (16).

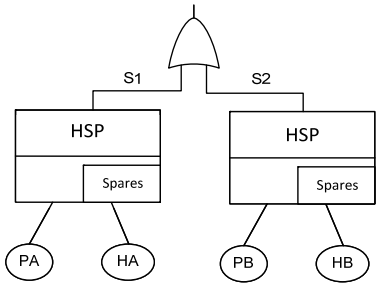

Fig. 8. DFT model of the cloud-based system (Phase 1)

$$R(t) = 1 - U_{OR}(t) = 1 - (U_{S1}(t) + (1 - U_{S1}(t)) * U_{S2}(t)) \tag{16}$$

where $U_{S1}(t)$ and $U_{S2}(t)$ are the unreliability functions of the subtrees $S1$ and $S2$, respectively. According to Eq. (7), $U_{S1}(t)$ and $U_{S2}(t)$ can be calculated as in Eq. (17) and Eq. (18), respectively. Note that Eqs. (17-18) have been simplified due to the assumed configurations, where $\lambda_{HA*} = \lambda_{PA}$ and $\lambda_{HB*} = \lambda_{PB}$.

$$U_{S1}(t) = 1 - R_{S1}(t) = 1 - (1 + \tfrac{\lambda_{PA}}{\lambda_{HA}})e^{-\lambda_{PA}t} + (\tfrac{\lambda_{PA}}{\lambda_{HA}})e^{-(\lambda_{PA}+\lambda_{HA})t} \tag{17}$$

$$U_{S2}(t) = 1 - R_{S2}(t) = 1 - (1 + \tfrac{\lambda_{PB}}{\lambda_{HB}})e^{-\lambda_{PB}t} + (\tfrac{\lambda_{PB}}{\lambda_{HB}})e^{-(\lambda_{PB}+\lambda_{HB})t} \tag{18}$$

In Phase 2, we consider both of the scenarios mentioned in the end of Section 4.3, so their impacts on system reliability as well as their consequent rejuvenation schedules can be compared. Figure 9 shows the DFT model of the cloud-based system in Phase 2 based on Scenario 1. For the same reason as in Phase 1, the system reliability can be calculated as in Eq. (19). According to Eq. (15), $U_{S3}(t)$ and $U_{S4}(t)$ can be calculated as in Eq. (20) and Eq. (21), respectively.

$$R(t) = 1 - U_{OR}(t) = 1 - (U_{S3}(t) + (1 - U_{S3}(t)) * U_{S4}(t)) \tag{19}$$

$$U_{S3}(t) = U_{S1}(t) * U_{S1'}(t) \tag{20}$$

$$U_{S4}(t) = U_{S2}(t) * U_{S2'}(t) \tag{21}$$

Note that in Eqs. (20-21), $U_{S1}(t)$, $U_{S1'}(t)$, $U_{S2}(t)$ and $U_{S2'}(t)$ can be calculated in a similar way as in Eqs. (17-18).

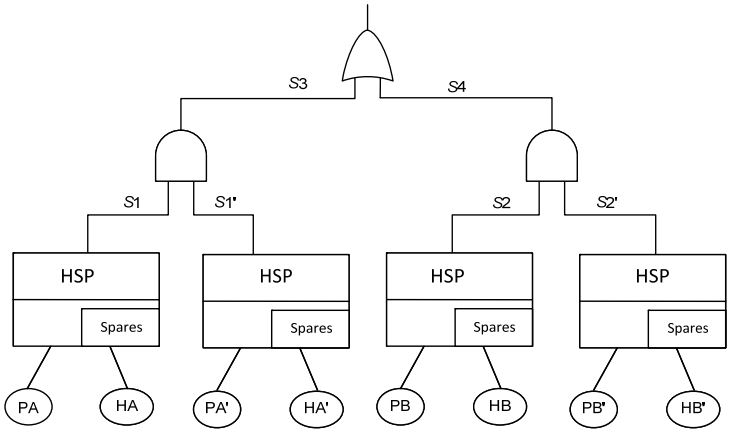

Fig. 9. DFT model of the cloud-based system in Phase 2 (Scenario 1)

The reliability analytical results for Scenario 1 are listed in Table 1. The table shows that the reliability threshold (0.99) is reached every 18 days. Hence, both the application and database servers are rejuvenated at the end of Phase 1. As Phase 2 has a 30-minute time duration, we calculate the system reliability at 5, 10, 20 and 30 minutes in Phase 2 to illustrate how system reliability may change during the rejuvenation process. From the table, we can see that the system reliability is kept very high during the transition. After 30 minutes, the newly deployed servers completely take over the system, and the servers to be rejuvenated are shut down. When this happens, the system returns to its initial state, and starts a new life cycle with a very high initial reliability. According to Table 1, we suggest that the system should be rejuvenated every 18 days in order to maintain the system reliability above the threshold.

By further looking into Table 1, we notice that when the system reliability reaches 0.99 after 18 days, the reliability of the database server subsystem is always lower than that of the application server subsystem. This suggests that we may first rejuvenate the most critical components with the lowest reliability (e.g., the database servers in this case study) without sacrificing the system reliability too much. Then we wait until the system reliability reaches the threshold again, and rejuvenate the application servers next, as they now become the most critical components. This is exactly what happens in the rejuvenation schedule of Scenario 2, where the application servers and the database servers are rejuvenated alternatively. Figure 10 shows the DFT model of the cloud-based system in Phase 2 for one of the two cases in Scenario 2, where only the database servers are rejuvenated. In this case, the system reliability can be calculated as in Eq. (22), and $U_{S1}(t)$ and $U_{S4}(t)$ can be calculated in a similar way as in Eq. (17) and Eq. (21), respectively.

Table 1. System reliability with software rejuvenation (Scenario 1)

Phase	Time (Days)	App Servers Reliability	DB Servers Reliability	System Reliability
1	0	1	1	1
	1	0.99998705	0.9999801	0.99996715026
	5	0.9996806	0.9995107	0.99919145628
	10	0.998745	0.998085	0.99683240333
	18	0.996044	0.994004	0.99007172018
2	18.0035	0.99999999999	0.99999999999	0.99999999999
	18.0069	0.99999999999	0.99999999999	0.99999999999
	18.0139	0.99999999999	0.99999999998	0.99999999997
	18.0208	0.99999999998	0.99999999994	0.99999999992
...	...	...	...	...
1	73	0.99998705	0.9999801	0.99996715026
	77	0.9996806	0.9995107	0.99919145628
	82	0.998745	0.998085	0.99683240333
	90	0.996044	0.994004	0.99007172018
2	90.0035	0.99999999999	0.99999999999	0.99999999999
	90.0069	0.99999999999	0.99999999999	0.99999999999
	90.0139	0.99999999999	0.99999999998	0.99999999997
	90.0208	0.99999999998	0.99999999994	0.99999999992
1	91	0.99998705	0.9999801	0.99996715026
	95	0.9996806	0.9995107	0.99919145628
	100	0.998745	0.998085	0.99683240333
	108	0.996044	0.994004	0.99007172018
2	108.0035	0.99999999999	0.99999999999	0.99999999999
	108.0069	0.99999999999	0.99999999999	0.99999999999
	108.0139	0.99999999999	0.99999999998	0.99999999997
	108.0208	0.99999999998	0.99999999994	0.99999999992
1	109	0.99998705	0.9999801	0.99996715026
	113	0.9996806	0.9995107	0.99919145628
	118	0.998745	0.998085	0.99683240333

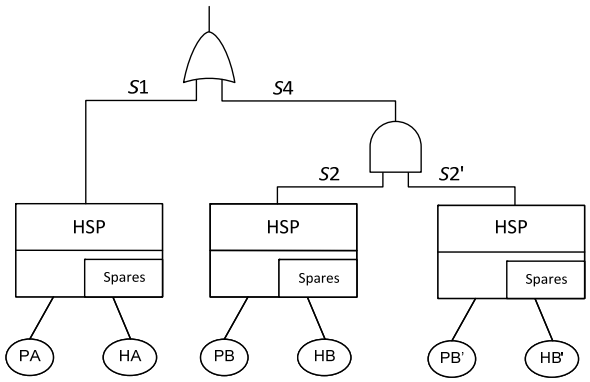

Fig. 10. DFT model of the cloud-based system in Phase 2 (Scenario 2, rejuvenate database servers only)

$$R(t) = 1 - U_{OR}(t) = 1 - (U_{S1}(t) + (1 - U_{S1}(t)) * U_{S4}(t)) \tag{22}$$

The system reliability for the other case in Scenario 2, where only the application servers are rejuvenated, can be calculated in a similar way. Table 2 shows the reliability analytical results for Scenario 2. At the end of each Phase 1, the server subsystem with its reliability marked by "=>" is the one to be rejuvenated. For example, after 18 days, the database servers are rejuvenated, and after 27 days, the application servers are rejuvenated.

Table 2. System reliability with software rejuvenation (Scenario 2)

Phase	Time (Days)	App Servers Reliability	DB Servers Reliability	System Reliability
1	0	1	1	1
	1	0.99998705	0.9999801	0.99996715026
	5	0.9996806	0.9995107	0.99919145628
	10	0.998745	0.998085	0.99683240333
	18	0.996044	=> 0.994004	0.99007172018
2	18.0035	0.9960425	0.99999999999	0.99604249999
	18.0069	0.996041	0.99999999999	0.99600409999
	18.0139	0.996038	0.99999999998	0.99603799998
	18.0208	0.996035	0.99999999994	0.99603499994
1	20	0.995150	0.99992069	0.99507107465
	25	0.992552	0.9990492	0.99160828156
	27	=> 0.9914	0.998442	0.99000000000
2	27.0035	0.99999999999	0.998441	0.99844099999
	27.0069	0.99999999999	0.998439	0.99843899999
	27.0139	0.99999999999	0.998437	0.99843699999
	27.0208	0.99999999998	0.998435	0.99843499998
1	30	0.9998842	0.997265	0.99714951671
	35	0.999109	0.994628	0.99374178645
	39	0.998205	=> 0.991942	0.99016146411
2	39.0035	0.998204	0.99999999999	0.99820399999
	39.0069	0.998202	0.99999999999	0.99820199999
	39.0139	0.99820	0.99999999997	0.99819999997
	39.0208	0.998199	0.99999999993	0.99819899993
...	...	...	...	...
1	85	0.998486	0.99992069	0.99840681007
	90	0.996853	0.9990492	0.99590519217
	95	0.994671	0.997265	0.99195057481
	96	0.994172	0.996804	0.99099462629
	97	=> 0.993652	0.99631	0.99000000000
2	97.0035	0.99999999999	0.99630	0.99680299999
	97.0069	0.99999999999	0.996298	0.99680099999
	97.0139	0.99999999998	0.996294	0.99679799998
	97.0208	0.99999999996	0.996291	0.99679399996
1	100	0.9998842	0.994628	0.99451282208
	105	0.9991090	=> 0.991194	0.99031084615
2	105.0035	0.9991902	0.99999999999	0.99874399999
	105.0069	0.9991895	0.99999999999	0.99874199999
	105.0139	0.9991882	0.99999999997	0.99873999997
	105.0208	0.9991868	0.99999999992	0.99873799992
1	110	0.9979	0.9995107	0.99741172753
	115	0.996044	0.998085	0.99413657574
	119	=> 0.994172	0.99631	0.99050350532

We now illustrate the suggested rejuvenation schedules for both Scenario 1 and Scenario 2 as in Fig. 11. In the figure, the start of rejuvenation is indicated by a sudden increment of the system reliability. By comparing the two rejuvenation schedules, we can see that during 119 days, Scenario 1 has 6 rejuvenation processes that require us to rejuvenate both of the application and database servers. On the other hand, Scenario 2 has 9 rejuvenation processes that only require us to rejuvenate either the application servers or the database servers each time. It is easy to see that Scenario 2 results in less management of the servers in order to keep the system reliability above the 0.99 threshold during the 119 days. Suppose the rejuvenation of the application servers has the same cost as the that of the database servers, by using the rejuvenation schedule defined in Scenario 2, the cost can be reduced by (6*2 - 9)/(6*2) = 25%, comparing to the rejuvenation schedule used in Scenario 1.

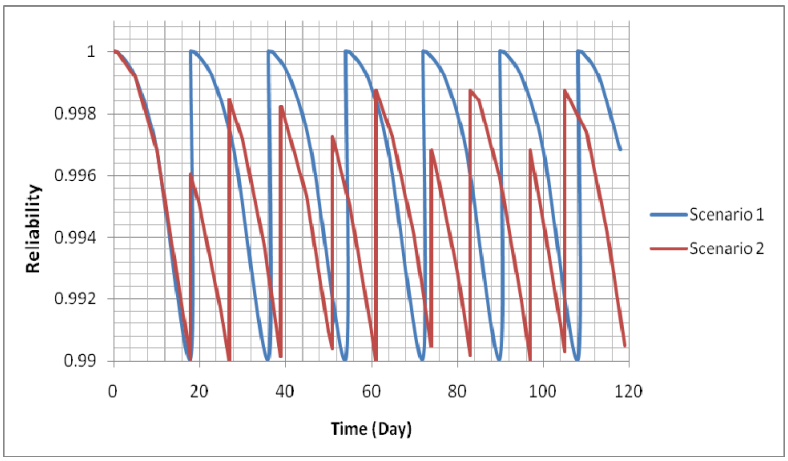

Fig. 11. Rejuvenation scheduling for the cloud-based system (Scenario 1 vs. Scenario 2)

6. Conclusions and Future Work

In this paper, we propose a reliability-based approach to establishing cost-effective software rejuvenation schedules for cloud-based systems. The system requires the usage of hot spare components during normal running time, and cold spare components during the rejuvenation process in order to keep the system reliability above a predefined threshold. By modeling the reliability of a cloud-based system using DFT, we are able to derive the reliability function for each software component as well as the whole system. We define two phases for the software rejuvenation, and discuss about two scenarios of the rejuvenation process in Phase 2. The analytical results of our case study show that Scenario 2 is more cost-effective than Scenario 1.

For future work, we will extend our current work for components with non-constant failure rates. We will adopt a measurement-based approach to collecting empirical data in

order to determine the *pdf*s of the major software components, the reliability of which is affected by software aging. Software tools will be developed for modeling and analyzing the reliability of cloud-based systems, as well as deriving effective rejuvenation schedules. In addition, we will expand and apply our proposed approach in more complex cloud environments, such as cloud-based systems using Amazon Web Services (AWS). Comparative analysis of system performance will be conducted for our proposed approach as well as existing fault-tolerant strategies that improve the reliability of cloud applications [26]. Finally, we envision modeling and analyzing cloud-based systems with active standby spare components, which can share workload with the primary ones [27], as a future, and more ambitious research direction.

Acknowledgments

We thank all anonymous referees for the careful review of this paper, and the many useful comments and suggestions that greatly helped us to improve the presentation and the quality of the paper.

References

1. K. V. Vishwanath and N. Nagappan, Characterizing cloud computing hardware reliability, in *Proc. of the 1st ACM symposium on Cloud computing (SoCC'10)*, Indianapolis, IN, USA, June 10-11, 2010, pp. 193-204.
2. D. Fitch and H. Xu, A RAID-based secure and fault-tolerant model for cloud information storage, *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* **23**(5) (2013) 627-654.
3. M. Rausand and A. Høyland, *System Reliability Theory: Models, Statistical Methods, and Applications*, Second Edition, Hoboken, New Jersey, USA, John Wiley & Sons, Inc., 2004.
4. H. Pham, *System Software Reliability*, Springer Series in Reliability Engineering, Springer-Verlag London, 2006.
5. A. Somani and N. Vaidya, Understanding fault tolerance and reliability, *IEEE Computer* **30**(4) (1997) 45-50.
6. E. Marshall, Fatal error: how patriot overlooked a scud, *Science* **255**(5050) (1992) 1347.
7. M. Grotte, R. Matias and K. S. Trivedi, The fundamentals of software aging, in *Proc. of the 1st International Workshop on Software Aging and Rejuvenation (WoSAR 2008)*, ISSRE, Seattle, WA, USA, November 11-14, 2008, pp. 1-6.
8. Y. Huang, C. Kintala, N. Kolettis and N. Fulton, Software rejuvenation: analysis, module and applications, in *Proc. of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS '95),* Pasadena, CA, USA, June 27-30, 1995, pp. 381-390.
9. J. Rahme and H. Xu, Reliability-based software rejuvenation scheduling for cloud-based systems, in *Proc. of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE 2015)*, Pittsburgh, USA, July 6-8, 2015, pp. 298-303.
10. V. Castelli, R.E. Harper and P. Heidelberger, *et al.*, Proactive management of software aging, *IBM Journal of Research and Development* **45**(2) (2001) 311-332.
11. L. Jiang and G. Xu, Modeling and analysis of software aging and software failure, *Journal of Systems and Software* **80**(4) (2007) 590-595.
12. A. Bobbio, M. Sereno and C. Anglano, Fine grained software degradation models for optimal rejuvenation policies, *Performance Evaluation* **46**(1) (2001) 45-62.

13. K. Vaidyanathan, D. Selvamuthu and K. S. Trivedi, Analysis of inspection-based preventive maintenance in operational software systems, in *Proc. of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002)*, Suita, Japan, October 13-16, 2002, pp. 286-295.
14. T. Dohi, K. Goseva-Popstojanova and K. S. Trivedi, Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule, in *Proc. of International Symposium on Dependable Computing*, Los Angeles, CA, USA, December 2000, pp. 77-84.
15. V. P. Koutras and A. N. Platis, Applying software rejuvenation in a two node cluster system for high availability, in *Proc. of the International Conference on Dependability of Computer Systems*, Szklarska, Poreba, May 25-27, 2006, pp. 175-182.
16. F. Machida, A. Andrzejak, R. Matias and E. Vicente, On the effectiveness of Mann-Kendall test for detection of software aging, in *Proc. of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Pasadena, CA, USA, November 4-7, 2013, pp. 269-274.
17. M. Grottke, L. Li, K. Vaidyanathan and K. S. Trivedi, Analysis of software aging in a web server, *IEEE Trans. on Reliability* **55**(3) (2006) 411-420.
18. J. Guo, Y. Ju, Y. Wang and X. Li, The prediction of software aging trend based on user intention, in *Proc. of the IEEE Youth Conference on Information Computing and Telecommunications (YC-ICT)*, Beijing, China, November 28-30, 2010, pp. 206-209.
19. D. Cotroneo, R. Natella and R. Pietrantuono, Is software aging related to software metrics? in *Proc. of the IEEE Second International Workshop on Software Aging and Rejuvenation (WoSAR)*, San Jose, CA, USA, November 2, 2010, pp. 1-6.
20. F. Machida, D. Kim and K. Trivedi, Modeling and analysis of software rejuvenation in a server virtualized system, in *Proc. of the IEEE Second International Workshop Software Aging and Rejuvenation (WoSAR)*, San Jose, CA, USA, November 2, 2010, pp. 1-6.
21. T. Thein, S.-D. Chi and J. S. Park, Availability modeling and analysis on virtualized clustering with rejuvenation, *International Journal of Computer Science and Network Security (IJCNS)* **8**(9) (2008) 72-80.
22. J. Barr, A. Narin and J. Varia, Building fault-tolerant applications on AWS, *Amazon Web Services (AWS)*, Amazon, October 2011, retrieved on July 15, 2015, from http://media.amazonwebservices.com/AWS_Building_Fault_Tolerant_Applications.pdf
23. H. Xu, L. Xing and R. Robidoux, DRBD: dynamic reliability block diagrams for system reliability modeling, *International Journal of Computers and Applications (IJCA)* **31**(2) (2009) 132-141.
24. R. Robidoux, H. Xu, L. Xing and M.C. Zhou, Automated modeling of dynamic reliability block diagrams using colored Petri nets, *IEEE Trans. on Systems, Man, and Cybernetics, Part A: Systems and Humans (SMC-A)* **40**(2) (2010) 337-351.
25. J. B. Dugan, S. J. Bavuso and M. A. Boyd, Dynamic fault-tree models for fault-tolerant computer systems, *IEEE Trans. on Reliability* **41**(3) (1992) 363-377.
26. M. Lu and H. Yu, A fault tolerant strategy in hybrid cloud based on QPN performance model, in *Proc. of the International Conference on the Information Science and Applications (ICISA)*, Pattaya, Thailand, June 24-26, 2013, pp. 1-7.
27. L. Huang and Q. Xu, Lifetime reliability for load-sharing redundant systems with arbitrary failure distributions, *IEEE Trans. on Reliability* **59**(2) (2010) 319-330.