Dependable and Reliable Cloud-Based Systems Using Multiple Software Spare Components

Jean Rahme and Haiping Xu
Computer and Information Science Department
University of Massachusetts Dartmouth, Dartmouth, MA 02747, USA
E-mail: {jrahme, hxu}@umassd.edu

Abstract—Cloud computing relies on a set of service components running on a service provider datacenter to achieve specific tasks. A trusted cloud-based software system is a highly dependable, reliable, available and predictable advanced computing system with guaranteed Quality of Service (QoS). Due to well-established studies and practices on hardware reliability, software faults have become the major factor of system failures in cloud-based systems. In this paper, we introduce a scheme of developing dependable and reliable cloud-based systems using multiple software spare components. We address the software-aging phenomenon in cloud computing, where the reliability of a software component decreases along the time. To counteract the software aging issue, we propose a mechanism to maintain the system reliability above a predefined safety threshold using software rejuvenation schedules. The calculation of system reliability is based on an extended Dynamic Fault Tree (DFT) model of cloud-based systems with Software SPare (SSP) gates. We verify our approach using Continuous Time Markov Chain (CTMC) for the case of constant failure rates, and provide a case study of a cloud-based system to show the detailed procedure as well as the feasibility of our approach.

Keywords—Software aging; hot software spare; cold software spare; software spare gate; reliability analysis; dynamic fault tree; software rejuvenation schedule

I. INTRODUCTION

As the cloud computing paradigm continues to grow along with the rapid computing technology advancement, cloud-based services are increasingly being used in many different areas such as healthcare, public transportation, mobile cloud computing and many more. A trusted cloud-based system is a highly dependable, reliable, available and predictable advanced computing system with guaranteed Quality of Service (QoS). The QoS of a computer-based system has been widely researched to maintain fault-tolerant hardware, secure, available and reliable software resources for client consumption. However, system outage of cloud-based systems is still common despite the well-established fault-tolerant techniques for hardware [1]. Software related faults of cloud-based systems due to the software-aging phenomenon [2] have become one of the major obstacles to achieving high fault tolerance and system reliability. Therefore, we were motivated to resolve the software aging related issues in cloud computing in order to maintain high dependability and reliability of cloud-based systems. In this work, we perform system reliability analysis from the perspective of Software Reliability Engineering (SRE). Early SRE focused on the analysis of software defects and bugs including Bohrbugs and Heisenbugs; while recently, the concept of software aging was introduced [3], taking into account the growing usage of cloud computing and the increasing workload that impacts the reliability of cloud-based systems. The software aging phenomenon is due to the degradation of system resources used by a software system until failure, which is caused by many factors such as memory bloating, memory leaks, data corruption, unreleased file-locks, unterminated threads, accumulation of round-off errors, and storage and space fragmentation [4]. To counteract the software aging problem, software rejuvenation has been proposed as a solution for achieving high fault tolerance in software-based systems [5]. Software rejuvenation can be done in many different ways, where the simplest one is to restart the application that causes the aging problem, or to reboot the whole system.

Correctly measuring the reliability of a cloud-based system is critical to avoid software failures due to the software-aging phenomenon. In this paper, we extend our former analytical-based approach to deriving the reliability function of a hot spare gate with a single hot standby spare [6]. In our new approach, multiple software spares are used for critical software components in cloud computing. The significance of our new approach is described as follows. First, as our former approach does not scale well for multiple hot spare parts, when a second hot spare is added into the system design, the analytical approach becomes non-trivial for the formalization of the analysis process. Second, it is useful and important to understand how a rejuvenation schedule may be affected by multiple hot software spares. We show the technical details for deriving the reliability function for a Software SPare (SSP) gate with two hot spare components, and use Continuous Time Markov Chain (CTMC) to verify the correctness of our approach for constant failure rates. A practical case study of a cloud-based system with two hot spares for two critical software components has been provided. In the case study, we assume a reliability threshold for triggering the software rejuvenation process, and based on the cloud-based system reliability analysis, we derive a software rejuvenation schedule to improve system reliability, dependability and availability.
II. RELATED WORK

Considerable research has been conducted on software aging and software rejuvenation to achieve high fault tolerance in software systems. There are mainly two categories of approaches to predicting software rejuvenation schedules, namely measurement-based and analytical-based approaches [7]. Measurement-based approach uses statistical analysis for the measured data of resource degradation that leads to software aging faults. A monitoring program collects the data, and analyzes them in order to estimate the degradation level. The rejuvenation process is triggered based on a predefined degradation threshold. Grottke et al. analyzed the resource degradation in a web server subject to injected workload [8]. The existence of monotonic trends was tested in time series, where these trends are indications of the software aging issues. Machida et al. detected software aging by applying Mann-Kendall test that is based on traces of computer system metrics [9]. Guo et al. established a trend prediction method to uncover software aging based on the quality of user requests [10]. Measurement-based approaches are feasible ways of predicting software aging, but they are quite inaccurate, and expensive in computational requirements due to the processing of large amounts of system data. Therefore, they are inefficient approaches in practical usage. However, when we use the time-to-failure distribution for data fitting and the calculation of system reliability, the estimated distribution from measurements can be useful in our proposed analytical-based approach [11].

On the other hand, in analytical-based approaches, we first have to assume failure time distributions for the components or the systems subject to software aging, and then schedule software rejuvenation processes at fixed interval based on the analytical results of the system reliability and availability. Bobbio et al. suggested a fine-grained software degradation model for optimal rejuvenation scheduling [12], by identifying the system current degradation level that outlines two different strategies of rejuvenation policies. Vaidyanathan et al. worked on an analytical model for software systems that uses inspection-based software rejuvenation [13]. They showed the advantages of inspection-based maintenance over non-inspection-based maintenance using Semi-Markov modeling. Koutras and Platis addressed a software rejuvenation technique for cluster systems, where rejuvenation can be carried out when node-deployed software starts to experience degradation, and thus an unscheduled reboot may be avoided [14]. Despite the fact that the above approaches introduced different models for software rejuvenation, they cannot be used to model dynamic behaviors such as sparing and dynamic relationships. Unlike the existing analytical-based approaches, our method studies the dynamic behaviors of software components in cloud-based software systems, namely, the standby software sparing components, and provides a novel analytical approach for reliability analysis.

Moreover, in the context of standby systems, there are four categories of evaluation methods for analyzing standby systems, namely simulations, state-space based methods, analytical/combinatorial approach, and numerical approach. As mentioned previously, simulation methods are expensive in terms of computations, and can only lead to approximate results [15]. Markov-based methods are state-space oriented [16], while non-Markovian models [17] are powerful in dynamic modeling. However, Markov-based models are limited to exponential failure distributions, and both of the approaches experience the state-space explosion problem when modeling large systems. Analytical approaches, such as minimal cut sets or sequences [18], and sequential decision diagrams [19] are limited to modeling complex behaviors with various time-to-failure distribution types. In our approach, we propose an extended DFT to model the reliability of cloud-based systems. We introduce an analytical-based approach to analyzing the extended DFT model for reliability calculation. Our approach does not suffer from the state-space explosion problem as it is compositional, where a DFT is decomposed into subtrees, and the system reliability is calculated by joining the reliabilities of the subtrees. Finally, numerical methods have been used as an iterative way for analyzing various designs of standby systems with a discrete approximation of time-to-failure distributions [20]. It is potential that a numerical method as demonstrated in previous work could be useful in our proposed analytical-based approach for estimating the time-to-failure distribution.

Finally, there is also some previous work on virtualized datacenter and cloud-based systems. Machida et al. proposed an availability model for virtualized systems with time-based rejuvenation using Petri-nets and a gradient search method [21]. Thein et al. modeled the availability of application servers, and they showed the high-availability cluster failover, combining virtualization and software rejuvenation [22]. However, none of the above approaches addressed explicitly the reliability analysis for software rejuvenation scheduling. In our approach, we analyze system reliability using an extended DFT model and use the proposed analytical approach to estimate rejuvenation schedules that satisfy predefined reliability requirements for cloud-based systems.

III. REJUVENATION IN THE CLOUD USING SOFTWARE SPARES

Virtualization allows multiple clients to share a physical machine’s resources using virtual machines (VM). To maintain high fault tolerance of a cloud-based system subject to software aging, we employ software rejuvenation and standby sparing for software redundancy to ensure service continuity. Different from physical machines, VMs are stored as images, which can be easily created, managed, and destroyed, making them very suitable for disaster recovery and disaster prevention. Comparing to hardware spares, using sparing VMs for disaster prevention and software rejuvenation could be a very inexpensive and effective way to restore the high performance of a cloud-based software system.

To achieve a reliable and zero-downtime rejuvenation, we define two types of VM spares, namely Hot Software Spare (HSS) and Cold Software Spare (CSS). In the context of cloud-based systems, an HSS is a hot standby VM instance that can be instantly available when a primary component fails. Despite the fact that an HSS is running alongside a primary component, it is not sharing any workload or processing any requests. Therefore, an HSS is operated using much less CPU power, but can be scaled automatically to meet the workload requirements when a primary component fails.
Critical data in an HSS is mirrored in near real time from the primary VM instance, e.g., in the range of 200 µs, to ensure high fault tolerance. The failure rate of an HSS is much less than that of a primary component as an HSS is not subject to aging-related bugs. This makes a software-defined HSS differ significantly from a hardware-based Hot Spare (HSP) because, with physical wearout, an HSP may have the same failure rate as a primary hardware component. On the other hand, a CSS refers to a software component that is available as an image of a VM, and can be replicated and deployed as a primary component or a HSS component. As an inactive VM instance, a CSS is mirrored for its critical data based on a specified schedule with most of the time being cold standby. Therefore, the reliability of a CSS is nearly perfect, which can be reasonably assumed never to fail. The recovery time using a CSS is usually in the range of minutes up to two hours; while the cost of a CSS is its storage and very little CPU resource consumption. A CSS can be rapidly deployed, which makes it quite different from a hardware-based Cold Spare (CSS) that is much expensive and requires manual configuration when a primary one fails.

Software rejuvenation techniques have been used to prevent the occurrence of aging-related software failures by proactively resetting a system’s internal state to its initial condition. In this work, we adopt an easy way of software rejuvenation by rebooting the system according to a defined schedule. In cloud computing, we can start a new VM to replace an old one that has demonstrated unsatisfactory system performance. To render the fault tolerant of the critical components and minimize the frequency of the rejuvenation events, each critical primary component is equipped by at least two HSSs and one CSS. The only CSS is needed for the rejuvenation process – it can be replicated for all currently deployed software components including the primary one and the HSSs. A newly deployed component must wait until the old ones have finished processing their remaining requests before they can be destroyed.

In our approach, a rejuvenation process is triggered when the reliability of a system component or the whole system reaches a predefined threshold. Similar to [6], we assume the rejuvenation process (Phase 1) takes about 30 minutes, with sufficient time to start a CSS and complete all remaining requests before Phase 2 starts. As a CSS never fails, we only consider the primary component and its HSSs when calculating the system reliability. In addition, two scenarios are investigated for the rejuvenation procedure. One scenario, called system-specific rejuvenation, is to rejuvenate the whole system when the system reliability reaches a threshold. The second scenario is a component-specific one, in which the critical component with the lowest reliability is rejuvenated when the system reliability reaches a threshold. As shown in a case study, the component-specific rejuvenation demonstrates certain advantages over the system-specific approach.

IV. RELIABILITY MODELING AND ANALYSIS

Dynamic Fault Tree (DFT) extends the concept of static fault tree and introduces new modeling capabilities for spare components, functional dependency, and failure sequence dependency. In this paper, we further extend DFT for modeling software spare components in cloud-based systems with software aging phenomenon.

A. SSP Gate for Cloud-Based Systems with Two Hot Spares

Figure 1 shows a SSP gate with one primary component and two HSSs. The primary component is initially powered on, but when it fails, it is replaced by an alternate HSS following an enumeration sequence. Therefore, a SSP gate fails only when the primary component and the alternate HSS components fail. Suppose the constant failure rates of components $P$, $H_1$, and $H_2$ are $\lambda_P$, $\lambda_{H1}$, and $\lambda_{H2}$, respectively. When $P$ fails, $H_1$ takes the lead to replace $P$ as $H_1^*$, with $\lambda_{H1*} \geq \lambda_{H1}$ due to the software aging phenomenon, when it takes the full workload. The same thing happens to $H_2$ when $H_1^*$ fails – $H_2$ replaces $H_1^*$ as $H_2^*$ with $\lambda_{H2*} \geq \lambda_{H2}$. Note that $\lambda_{H1*}$ and $\lambda_{H2*}$ do not have to be equal because $P$ and $H$ may have different configurations. In addition, we designate $t_1$, $t_2$, and $t_3$ as the time to failure of $P$, $H_1$ and $H_2$, respectively.

To derive the reliability function of a SSP gate with two hot spares, we identify all the possible events when a SSP gate fails according to component failure sequence. We denote the event “component $X$ fails before component $Y$” as $X < Y$, and summarize six disjoint events $e_i$ where $1 \leq i \leq 6$, as in Fig. 2.

![Fig. 1. An SSP gate with a primary component and two HSSs](image)

![Fig. 2. Six events for the failure of an SSP gate with two HSSs](image)
Let event $A$ be the failure of an SSP gate at time $t$. We can calculate the probability of event $A$ as in Eq. (1):

$$
Pr(A) = \sum_{i=1}^{6} Pr(A \mid e_i) \cdot Pr(e_i) = \sum_{i=1}^{6} Pr(A \cap e_i)
$$

(1)

It is worth noting that when event $e_i$ happens, the SSP gate also fails. Therefore, event $A$ always happens with some event $e_i$. Thus, Eq. (1) can be simplified as in Eq. (2).

$$
Pr(A) = \sum_{i=1}^{6} Pr(e_i)
$$

(2)

**Event $e_i$:** $P$ fails before $H_1$, and $H_1$ fails before $H_2$, denoted as $P \prec H_1 \prec H_2$. In this case, it is guaranteed that $H_1$ does not fail during $(0, \tau_1)$, and $H_2$ does not fail during $(0, \tau_2)$. After $P$ fails, $H_1$ takes over the workload and becomes $H_1^*$, also after $H_1^*$ fails, $H_2$ takes over the workload and becomes $H_2^*$. Intuitively, the unreliability function $U(t)$ of the SSP gate, i.e., the probability that the SSP gate fails during $(0, t)$, can be calculated as in Eq. (3).

$$
Pr_{H_1^*, \text{actual}}(T \leq t) = \frac{1}{2} \int_{0}^{\tau_1} \int_{t_1}^{\tau_1} \int_{t_2}^{\tau_2} \lambda_{H_1^*} e^{-\lambda_{H_1^*} t} \cdot \lambda_{H_1^*} e^{-\lambda_{H_1^*} t} \cdot \lambda_{H_1^*} e^{-\lambda_{H_1^*} t} \cdot dt_1 dt_2 dt_3
$$

(3)

However, Eq. (3) only works when $\lambda_{H_1} = \lambda_{H_2}$. As shown in previous work [6], when $\lambda_{H_1} > \lambda_{H_2}$, the integration of the probability density function (pdf) of $H_2^*$ from $t_1$ to $t$ does not give the correct unreliability of the component at time $t$, as it incorrectly assumes that component $H_1$ behaves as $H_2^*$ starting from time 0. Since the component actually behaves as $H_1$ during $(0, \tau_1)$, the unreliability of $H_2^*$ at time $t$ equals the unreliability of $H_1^*$ at $t_1$ rather than the unreliability calculated by the integration of the pdf of $H_2^*$ from $t_1$ to $t_3$. This is to ensure the unreliability continuity for $H_1$ before and after it serves as a primary component $H_1^*$. By calculating a new starting integration time $t_{H_1^*}$ for $H_1^*$, we take into consideration that $t_3$, originally the failure of component $H_1$, is shifted to the left by $(\tau_1-t_{H_1^*})$. As a result, when we consider the failure of $H_1^*$, we must add $(\tau_1-t_{H_1^*})$ to $t_3$ since $H_2^*$ is activated based on the original non-shifted failure time variable $t_3$ of $H_1$. Therefore, the value of $t_3$ after the adjustment is given as $t_{3\text{actual}} = t_{3\text{shifted}} + (t_1-t_{H_1^*}) = t_{3\text{shifted}} + (t_1-t_{H_1^*})$, as shown in Fig. 3. As a rule of thumb, in the case of $P \prec H_1 \prec H_2 \prec H_0$, where $i > 1$, $t_3$ does not get shifted since it is the failure time of $P$, and $P$ always acts as a primary component, when a component $H_1^*$ acts as a primary one, its actual time to failure equals $t_{i+1\text{shifted}}(t_i-t_{H_1^*})$. This observation and adjustment is critical for yielding the correct reliability function.

Hot spare $H_2$ or the first HSS has been studied in previous work [6] yielding $t_{H_2} = (\lambda_{H_1}/\lambda_{H_2}) t_1$. In regards to the second HSS $H_2$, it is guaranteed that $H_2$ does not fail during $(0, \tau_2)$. After $H_1^*$ fails, $H_2$ takes over the workload and becomes $H_2^*$. Since the component actually behaves as $H_2$ during $(0, \tau_2)$, the unreliability of $H_2^*$ at time $t_2$ equals the unreliability of $H_2$ at $t_2$ rather than the unreliability calculated by the integration of the pdf of $H_2^*$ from 0 to $t_2$. This requires us to calculate a new starting integration time $t_{H_2^*}$ for $H_2^*$ such that the unreliability of $H_2^*$ at $t_{H_2^*}$ is equal to the unreliability of $H_2$ at $t_2$. As the pdfs of $H_2$ and $H_2^*$ are $f(t) = \lambda_{H_2} e^{-\lambda_{H_2} t}$ and $f(t) = \lambda_{H_2^*} e^{-\lambda_{H_2^*} t}$, respectively, such a relationship between $H_2$ and $H_2^*$ can be described as in Eq. (4), taking into account the adjustment of $t_2$, i.e., the time to failure of $H_2^*$.

$$
\int_{0}^{\tau_2} \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} dt_2 = \int_{0}^{\tau_2} \lambda_{H_2} e^{-\lambda_{H_2} t} dt_2
$$

(4)

Solving Eq. (4), we have $\tau_{H_2^*} = \frac{\lambda_{H_2}}{\lambda_{H_2^*}} (\tau_2 + (\tau_2 - \tau_{H_2^*}))$. Since $H_2^*$ fails during a period of time $(t, t_2)$, the integration range for $H_2^*$ now becomes $[t_{H_2^*}, t_2]$, as illustrated in Fig. 3. The probability of the event $P \prec H_1 \prec H_2$, i.e., $Pr(e_i)$, can be calculated as in Eq. (5).

$$
Pr_{H_1^*, H_2^*, \text{actual}}(T \leq t) = \int_{0}^{\tau_2} \int_{t_{H_2^*}}^{\tau_2} \int_{t}^{\tau} \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} dt_2 \cdot \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} \cdot \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} dt_3 dt_4 dt_5
$$

(5)

**Event $e_3$:** $P \prec H_2 \prec H_1$, this is where $P$ fails first then $H_1^*$ fails as a spare before $H_1^*$ fails. The failure of $H_2$ is independent of $H_2^*$, and the failure of $H_1^*$ depends on $P$ failure but not on $H_2^*$’s failure. The integration of $H_1^*$ requires computing $\tau_{H_1^*}$, which is based on $t_1$ by moving the integration limit from $t_{H_1^*}$ to $t_{H_1^*} + t_1$ - $t_{H_1^*}$, resulting in Eq. (7).

$$
Pr_{H_1^*, H_2^*, \text{actual}}(T \leq t) = \int_{0}^{\tau_2} \int_{t_{H_2^*}}^{\tau_2} \int_{t}^{\tau} \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} dt_2 \cdot \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} \cdot \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} dt_3 dt_4 dt_5
$$

(7)

**Event $e_4$:** $H_1 \prec H_2 \prec P$, $H_1$ and $H_2$ fail as spares before $P$ fails, similar to one spare SSP gate, where it is guaranteed that $P$ does not fail during $(0, \tau_1)$. The probability that the SSP gate fails during $(0, t)$ can be calculated as in Eq. (9).

$$
Pr_{H_1^*, H_2^*, \text{actual}}^P(T \leq t) = \int_{0}^{\tau_2} \int_{t_{H_2^*}}^{\tau_2} \int_{t}^{\tau} \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} dt_2 \cdot \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} \cdot \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} dt_3 dt_4 dt_5
$$

(9)

**Event $e_5$:** $H_1 \prec P \prec H_2$, similar to event $e_3$, this is where $H_1$ fails first as a spare, then $P$ and $H_2$ as $H_2^*$. Note that the complexity is similar to one spare SSP gate $P \prec H$. The probability that the SSP gate fails is calculated as in Eq. (8).

$$
Pr_{H_1^*, H_2^*, \text{actual}}(T \leq t) = \int_{0}^{\tau_2} \int_{t_{H_2^*}}^{\tau_2} \int_{t}^{\tau} \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} dt_2 \cdot \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} \cdot \lambda_{H_2^*} e^{-\lambda_{H_2^*} t} dt_3 dt_4 dt_5
$$

(8)

**Event $e_6$:** $H_2 \prec H_1 \prec P$, similar to event $e_6$, $H_2$ and $H_1$ fail as spares before $P$ fails, similar to one spare SSP gate, where it is guaranteed that $P$ does not fail during $(0, \tau_2)$. The
probability that the SSP gate fails during \((0, t]\) can be calculated as in Eq. (11).

\[
Pr_{f}(T \leq t) = \int_{0}^{t} \frac{1}{\lambda} \left[ e^{-\lambda t} \right] dT
\]

Based on Eq. (2), the unreliability function of the SSP gate with two HSSs is given in Eq. (12) given that the reliability function is \(R(t) = 1 - U(t)\).

\[
U(t) = Pr(T \geq t) = Pr(T \geq t_1) + Pr(T \geq t_2) + Pr(T \geq t_1) + Pr(T \geq t_2) + Pr(T \geq t_1) + Pr(T \geq t_2)
\]

**B. Reliability Function Verification Using CTMC**

We use a CTMC model to formally verify the correctness of the reliability function \(R(t)\) derived in the previous section. Fig. 4 shows the CTMC model corresponding to the SSP gate with two HSSs illustrated in Fig. 1.

![Fig. 4. The CTMC model of the SSP gate in Fig. 1](image)

There are 8 states in the model, denoted as \(PH_1H_2, H_1H_2, PH_3, PH_4, H_1H_2, H_2H_1, H_3H_4, P, \) and Failure. Each state holds the name of the surviving components, except the Failure state, which is the unavailable state. The reliability of the SSP gate is the sum of the probability of being in all available states, namely State 1 to State 7. Let \(P(t)\) be the probability of the system in state \(i\) at time \(t\), where \(1 \leq i \leq 8\), and \(P(0) = \prod_{j=0}^{i} (X(t) = j)\) be the incremental transition probability with random variable \(X(t)\). The matrix \([P_i(0)dt)]\) defined in Eq. (13), where \(1 \leq i, j \leq 8\), is the incremental one-step transition matrix of the CTMC defined in Fig. 4.

\[
\begin{bmatrix}
1-\lambda_1 dt & \lambda_1 dt & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1-\lambda_2 dt & \lambda_2 dt & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1-\lambda_3 dt & \lambda_3 dt & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1-\lambda_4 dt & \lambda_4 dt & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1-\lambda_5 dt & \lambda_5 dt & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1-\lambda_6 dt & \lambda_6 dt & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1-\lambda_7 dt & \lambda_7 dt \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

The transition matrix is a stochastic matrix with each row sums to 1, and it defines the probability for each state either remaining (when \(i = j\)) or transitioning to a different state (when \(i \neq j\)) 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 Eq. (13), we can derive the following relations as in Eqs. (14.1-14.7).

\[
P_1(t+dt) = (1-\lambda_1 + \lambda_2 + \lambda_3) dt P_1(t) + \lambda_1 dt P_2(t) + (1-\lambda_2 + \lambda_3 + \lambda_4) dt P_2(t)
\]

\[
P_2(t+dt) = \lambda_2 dt P_1(t) + (1-\lambda_2 + \lambda_3 + \lambda_4) dt P_2(t)
\]

\[
P_3(t+dt) = \lambda_3 dt P_1(t) + (1-\lambda_2 + \lambda_3 + \lambda_4) dt P_2(t)
\]

\[
P_4(t+dt) = \lambda_4 dt P_1(t) + (1-\lambda_3 + \lambda_4 + \lambda_5) dt P_2(t)
\]

\[
P_5(t+dt) = \lambda_5 dt P_3(t) + \lambda_4 dt P_2(t) + (1-\lambda_3 + \lambda_4 + \lambda_5) dt P_5(t)
\]

\[
P_6(t+dt) = \lambda_5 dt P_3(t) + \lambda_4 dt P_2(t) + (1-\lambda_3 + \lambda_4 + \lambda_5) dt P_5(t)
\]

We derive a set of linear first-order differential equations as in Eqs. (15.1-15.7), which are state equations of the CTMC model assuming the initial probabilities \(P_i(0) = 1\), and \(P_{i+1}(0) = P_{i}(0) = P_{i+2}(0) = P_{i+3}(0) = P_{i+4}(0) = P_{i+5}(0) = 0\).

\[
P_1(t+dt) = (1-\lambda_2 + \lambda_3 + \lambda_4) dt P_1(t) + \lambda_1 dt P_2(t) + (1-\lambda_2 + \lambda_3 + \lambda_4) dt P_2(t)
\]

\[
P_2(t+dt) = \lambda_2 dt P_1(t) + (1-\lambda_2 + \lambda_3 + \lambda_4) dt P_2(t)
\]

\[
P_3(t+dt) = \lambda_3 dt P_1(t) + (1-\lambda_2 + \lambda_3 + \lambda_4) dt P_2(t)
\]

\[
P_4(t+dt) = \lambda_4 dt P_1(t) + (1-\lambda_3 + \lambda_4 + \lambda_5) dt P_5(t)
\]

\[
P_5(t+dt) = \lambda_5 dt P_3(t) + \lambda_4 dt P_2(t) + (1-\lambda_3 + \lambda_4 + \lambda_5) dt P_5(t)
\]

\[
P_6(t+dt) = \lambda_5 dt P_3(t) + \lambda_4 dt P_2(t) + (1-\lambda_3 + \lambda_4 + \lambda_5) dt P_5(t)
\]

Using Laplace transformation to both sides of Eqs. (15.1-15.7) to derive Eqs. (16.1-16.7).

\[
sP_i(s) - P_i(0) = -(\lambda_2 + \lambda_3 + \lambda_4) P_i(s)
\]

\[
sP_i(s) - P_i(0) = \lambda_2 P_i(s) - (\lambda_2 + \lambda_3) P_i(s)
\]

\[
sP_i(s) - P_i(0) = \lambda_3 P_i(s) - (\lambda_2 + \lambda_3 + \lambda_4) P_i(s)
\]

\[
sP_i(s) - P_i(0) = \lambda_4 P_i(s) - (\lambda_3 + \lambda_4 + \lambda_5) P_i(s)
\]

\[
sP_i(s) - P_i(0) = \lambda_5 P_i(s) - (\lambda_4 + \lambda_5 + \lambda_6) P_i(s)
\]

\[
sP_i(s) - P_i(0) = \lambda_6 P_i(s) - (\lambda_5 + \lambda_6 + \lambda_7) P_i(s)
\]

Substituting the initial probabilities \(P_i(0)\), where \(1 \leq i \leq 7\), into Eqs. (16.1-16.7), we can derive the equations for \(P_{i+1}(s)\), \(P_{i+2}(s)\), \(P_{i+3}(s)\), \(P_{i+4}(s)\), and \(P_{i+5}(s)\). By applying inverse Laplace transformation, we can solve the original linear first-order differential equations as follows.

\[
P_{i}(s) = \frac{1}{(s + \lambda_2 + \lambda_3 + \lambda_4)} \Rightarrow P_i(t) = e^{-0\cdot\lambda_2 + \lambda_3 + \lambda_4} t
\]

\[
P_{i}(s) = \frac{\lambda_2 P_{i+1}(s)}{(s + \lambda_3 + \lambda_4)} \Rightarrow P_{i+1}(t) = \frac{\lambda_2}{\lambda_2 + \lambda_3 + \lambda_4} e^{-0\cdot\lambda_3 + \lambda_2 + \lambda_4} t
\]

\[
P_{i}(s) = \frac{\lambda_3 P_{i+2}(s)}{(s + \lambda_4 + \lambda_5)} \Rightarrow P_{i+2}(t) = e^{-0\cdot\lambda_4 + \lambda_3 + \lambda_5} t
\]

\[
P_{i}(s) = \frac{\lambda_4 P_{i+3}(s)}{(s + \lambda_5 + \lambda_6)} \Rightarrow P_{i+3}(t) = e^{-0\cdot\lambda_5 + \lambda_4 + \lambda_6} t
\]

\[
P_{i}(s) = \frac{\lambda_5 P_{i+4}(s)}{(s + \lambda_6 + \lambda_7)} \Rightarrow P_{i+4}(t) = e^{-0\cdot\lambda_6 + \lambda_5 + \lambda_7} t
\]

\[
P_{i}(s) = \frac{\lambda_6 P_{i+5}(s)}{(s + \lambda_7 + \lambda_8)} \Rightarrow P_{i+5}(t) = e^{-0\cdot\lambda_7 + \lambda_6 + \lambda_8} t
\]
The four hot spare components set up for failure rates for the servers, where $\lambda_{PA} = 0.004$/day, $\lambda_{HB1} = \lambda_{HB2} = 0.003$/day, using the same failure rates as in previous work [6], so the obtained results can be readily compared.

The reliability function $R(t)$ from CTMC analysis is given as in Eq. (17).

$$R(t) = P_1(t) + P_2(t) + P_4(t) + P_6(t) + P_7(t) + P_8(t) \quad (17)$$

We compute the system reliability using the reliability function $R(t)$ from both the proposed approach and the CTMC approach presented in Eq. (12) and Eq. (17), respectively, in Table 1. The results show they are perfectly matched.

The case study also involves CSS components, namely CSA and CSB, which are used in the rejuvenation process. Note that a CSS is a stored image of a deployed VM instance that can be easily duplicated, thus only one CSS is needed for each of the primary and HSS components. In addition, since a CSS is stored as an image, its failure rate is considered to be 0. However, once a CSS component is duplicated and deployed, it will assume the failure rate of its corresponding role, either as a running primary component or as an HSS. The DFT model of the cloud-based software system for Phase 1 is shown in Fig. 6.

Since the system fails when either the application server or the database server fails, the two SSP gates are connected by an OR-gate. The reliability function of the OR-gate is derived using sum of disjoint product as in Eq. (18).

$$R(t) = 1 - U_{OR}(t) = 1 - (U_{S1}(t) + (1 - U_{S1}(t)) \cdot U_{S2}(t)) \quad (18)$$

where $U_{S1}(t)$ and $U_{S2}(t)$ are the unreliability functions of the subtrees S1 and S2 that can be calculated using Eq. (12). We consider both scenarios mentioned in Section 3 for Phase 2 analysis. Fig. 7 represents the DFT model of the cloud-based system in Phase 2 for Scenario 1.
Similar to Phase 1, we can analyze the DFT model for Phase 2 Scenario 1 by splitting it into subtree sections. Starting from bottom to top, the unreliabilities $U_{S1}(t)$, $U_{S1'}(t)$, $U_{S2}(t)$ and $U_{S2'}(t)$ can be derived using Eq. (12). $U_{S3}(t)$ and $U_{S4}(t)$ can be calculated using the sum of disjoint product method for AND-gate shown in Eqs. (19-20). Finally, the system reliability for the OR-gate is derived as in Eq. (18) for Phase 1.

\[
U_{S3}(t) = U_{S1}(t) \times U_{S1'}(t) \\
U_{S4}(t) = U_{S2}(t) \times U_{S2'}(t)
\]

Moving to Phase 2 Scenario 2, the DFT model of the system is illustrated in Fig. 8 for the subsystem rejuvenation of the application servers.

Using the same methodology for DFT analysis, we have the following subtrees $U_{S1}(t)$, $U_{S1'}(t)$, $U_{S2}(t)$ and $U_{S2'}(t)$ in the DFT model. The unreliabilities $U_{S1}(t)$, $U_{S1'}(t)$ and $U_{S2}(t)$ can be derived using Eq. (12). $U_{S3}(t)$ is calculated using the sum of disjoint product method for AND-gate shown in Eq. (19).

As we have shown how to derive the system reliability in both Phase 1 and Phase 2, including the two different scenarios, the next step is to show the difference and the impact of employing 2-HSSs vs. 1-HSS [6] in terms of reliability and rejuvenation scheduling in a cloud-based system. In addition, we study the impacts of using Scenario 1 vs. Scenario 2 for rejuvenation scheduling for a cloud-based system with multiple HSSs subject to software aging.

Figure 9 illustrates the details of the difference between the two cases based on Scenario 1. Note that 1-HSS results are formerly provided in [6]. From the figure, we can see that the system reliability is kept very high during the transition. According to Fig. 9, the reliability threshold for 2-HSSs is reached at 48 days, hence it is suggested that the system should be rejuvenated every 48 days under Scenario 1. On the other hand, we can also see that the system needs to be rejuvenated every 18 days with a single HSS usage. Comparing rejuvenation scheduling based on reliability analysis for both cases over the period of 120 days, we notice that the system with 2-HSSs only needs two rejuvenations (at 48 and 96 days), but it requires six rejuvenations with a single HSS for its critical component. Therefore, Scenario 1 with 2-HSSs results in $(6*2-6*2)/(6*2) = 66\%$ reduction in cost and management for software rejuvenation, while keeping the system above the same reliability threshold (0.99).

Figure 10 shows Scenario 2 for component-specific software rejuvenation. According to the figure, when the system reliability reaches the threshold in 48 days, the components with the lowest reliability, i.e., the database servers, are scheduled for rejuvenation first. The rejuvenation induces a partial spike in the reliability curve, and then the system reliability is continuously monitored until it reaches the threshold again at the 69th day. At this point, the application server components become the ones with the lowest reliability. As a result, there will be an alternation in rejuvenation process for the two subsystems. We can see three rejuvenations for Scenario 2 with 2-HSSs vs. nine rejuvenations for 1-HSS design. Therefore, Scenario 2 with 2-HSSs results in $(9-3)/(9) = 66\%$ reduction in cost and
management for software rejuvenation, while keeping the system above the same reliability threshold (0.99).

Figure 11 compares the two scenarios with two HSSs in 120 days. Scenario 1 has two rejuvenations that require us to rejuvenate both of the application and database servers. On the other hand, Scenario 2 has three rejuvenations that only require us to rejuvenate either the application servers or the database servers each time. Thus, by using Scenario 2, we can reduce the rejuvenation cost and management by \((2 \times 2 - 3)/4 = 25\%\) compared to the case of Scenario 1.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a reliability-based approach using two HSSs for critical components during normal running time in cloud-based software systems. We defined an extension of DFT, called SSP gate, which can be used to evaluate the reliability of a cloud-based system with multiple software spares for its critical components. Our approach has been verified using CTMC for constant failure rates. The case study showed that using the proposed approach, a rejuvenation schedule can be derived to maintain the system reliability of a trusted cloud-based software system with multiple software spare components above a certain level.

For future work, a measurement-based approach can be adopted for collecting empirical data relative to the software aging phenomenon, and then we can use data fitting technique to obtain the pdfs of the critical software components. Once the pdfs become available, they can be plugged into our proposed analytical approach to evaluate the system reliability and estimate the rejuvenation schedules based on the collected data. Finally, we envision modeling and analyzing cloud-based systems with active standby spare components, which can share workload with the primary ones, as a future, and more ambitious research direction.

REFERENCES


