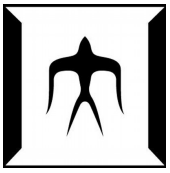
論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Modeling and analysis for design of dependable software systems and operations
著者(和文)	町田文雄
Author(English)	Fumio Machida
出典(和文)	学位:博士(学術), 学位授与機関:東京工業大学, 報告番号:甲第10902号, 授与年月日:2018年3月26日, 学位の種別:課程博士, 審査員:三好 直人,樺島 祥介,横田 治夫,DEFAGO XAVIER,中野 張
Citation(English)	Degree:Doctor (Academic), Conferring organization: Tokyo Institute of Technology, Report number:甲第10902号, Conferred date:2018/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Modeling and analysis for design of dependable software systems and operations

Fumio Machida

Thesis submitted for
the Degree of Doctor of Philosophy



Department of Mathematical and Computing Sciences
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

2018

Abstract

Information technology (IT) systems have become essential to our society as they are extensively used in our daily lives, economic activities, and social infrastructures. The effectiveness of software services delivered through IT systems directly impacts on human lives, organizational productivity, and social safety. Since malfunctions of such IT systems can cause serious problems, dependability and performability design of these systems is crucial. A fundamental challenge is the difficulties in dealing with various uncertainties in complex software systems, such as demand changes and component failures, that are inevitable during their operations. In this thesis, the applications of stochastic models are presented, particularly for dependability and performability system design and operation. Stochastic models are used to capture dynamic failure-recovery behaviors of systems as well as fault-tolerant configurations. Since recent IT systems are becoming more software-intensive owing to increased computing power, the impact of software faults and errors must be taken into account. In this thesis, dependability and performability design of software-based IT systems are discussed in three areas. In the first area, software aging problem, which is often observed in long-running software systems, is discussed. Against software aging, the effective countermeasures, software rejuvenation, and software life-extension, are evaluated. In the second area, the effectiveness of storage array configurations and their data management operations are evaluated. To achieve high data-availability with acceptable performance, optimal storage array configurations and data backup operations are analyzed. In the third area, the issue of cloud resource management to avoid performance problems due to resource contention is discussed. Virtual machine reallocation in a cloud data center and additional physical server procurement are considered. In all three cases, dynamic behaviors of systems are captured by stochastic models that are used to quantitatively evaluate the performance measures by using analytical solution techniques. Quantitative assessment enables design improvement and maintenance optimization toward achieving highly available and high-performance software-based systems.

Table of Contents

List of figures

List of tables

Acknowledgements

This thesis would have never been possible without extensive supports from my advisor, dissertation committee, colleagues, friends and my family.

First, I am deeply indebted to my advisor Prof. Naoto Miyoshi for giving me an opportunity to be a member of his research group. His guidance, encouragement and constant support during my Ph.D student period helped me a lot for continuing my research. His supportive and insightful advice have always added enormous values to the quality of my study. I am also grateful to him for keeping trust in me and my abilities to complete my research work.

I would like to thank the members of my thesis committee, Prof. Haruo Yokota, Prof. Xavier Defago, Prof. Yoshiyuki Kabashima, and Associate Prof. Yumiharu Nakano. Many suggestions and comments from different angles gave me an opportunity to think even more deeply about the value and impact of my research outcome. In particular, I am grateful to Prof. Haruo Yokota and Prof. Xavier Defago for giving me the comments to my presentation at the international conference in 2014.

All my research contributions have been based on numerous supports from my colleagues and collaborators. I would like to express my sincere gratitude to Prof. Kishor Trivedi at Duke University. He gave me the basis of research on dependable computing and put me in the right direction. Through the years of collaborations, we made several research outcomes and I have learned many things from him and his research group members. I would like to thank Dr. Ruofan Xia, especially for helping my work on data backup and storage system research when he was a student at Duke. I would also like to express my gratitude to my ex-colleagues: Dr. Yoshiharu Maeno, Dr. Jianwen Xiang, and Ms. Kumiko Tadano for collaborative research on software rejuvenation and software life-extension. It was one of the greatest periods that I worked with them in the same research group. I would also like to thank Prof. Paulo Maciel for kindly accepting the use of our co-authored survey report for software rejuvenation models in my thesis. Although this thesis does not cover all of my

previous publications before the period of the thesis, the content of the thesis heavily relied on the results of the previous studies conducted with my ex-collaborators. I would like to express my sincere gratitude to these ex-collaborators on my thesis topic: Prof. Victor Nicola, Prof. Jogesh Muppala, Prof. Artur Andrzejak, Prof. Rivalino Matias Jr., Prof. Poul Heegaard, Prof. Bjarne Helvik, Dr. DongSeong Kim, Dr. Javier Alonso, Dr. Xiaoyan Yin, Dr. Ermeson Andrade, Dr. Rubens Matos, Dr. Subrota Mondal, Dr. Diego Elias and many colleagues in NEC Corporation.

My research has been also supported by many friends and colleagues in the institute and company. I would have not been able to do this work without understandings of my superiors and colleagues in NEC Corporation. I would also like to express my special thanks to the members of Miyoshi's research group for assisting my participation of group activities. I am also grateful to Ms. Hiroko Ohmura for supporting some important administrative procedures.

Finally, I would like to thank my family for all their continuous supports and encouragements. My parents were always encouraging me with their best wishes. My wife, Junko Machida, was always cheering me up and stood by my side with faithful support. I deeply appreciate her efforts and patience. I was also cheered up by my son who is in her womb now and will be born in a month.

Chapter 1

Introduction

1.1. Background

Today's smart social infrastructures could not have been achieved without support from advanced information technology (IT) and network systems. Our safety and efficient economic activities highly depend on IT systems including cloud computing, mobile devices, social applications, various types of web services, and Internet of Things (IoT) systems. Utility services, such as power grids, water supply, and gas pipelines, are becoming interconnected via the Internet, and their statuses are continuously monitored by IT systems in operation centers or distributed edge devices. Enterprises and organizations are now in the midst of digital transformation through which their productivities and collaboration capabilities are revolutionized by the digitization of organizational activities, making full use of IT. In the medical and healthcare domain, hospital operation, personalized medicine, and healthcare monitoring are also being digitized, accordingly human lives and health conditions have become more dependent on IT systems.

As the use of IT systems has spread throughout society, dependability of such systems become crucial for users and organizations. Faulty systems could be catastrophic on our social infrastructures. For example, unavailability of utility services directly impacts on our lives in terms of electricity, water and gas supplies. In enterprises, service outage of IT systems, such as cloud computing services, might result in a huge economic loss. In hospitals, service interruption of medical services could lead to life-threatening consequences. To make our society more dependable, IT-supported services should be highly available such that even under faulty conditions, systems can provide continuous service with a guaranteed service

level for users.

Considering the dependability of IT systems, it is important to look at the recent trend in IT-system architecture, which tends to be highly software-intensive. Owing to the commoditization of computing hardware and increasing network capacity, computing resources can be easily acquired from cloud computing infrastructure at lower cost. On top of the commodity of servers, software plays a significant role in configuring different type of application services. In particular, server virtualization and containers are now becoming essential building blocks of flexible software systems such that their execution instances are easily created, deleted, and migrated among different hardware machines through application programmable interfaces (APIs). Software-defined network (SDN) and network function virtualization (NFV) are representative examples of softwarelization, as traditional network equipment is replaced with software-based network functions deployed on commodity hardware. Softwarelization is progressing in many areas, e.g., software-defined storage, software-defined infrastructure, software-defined security. Software-intensive IT systems have become more complex because of the increased number of involved software components developed by different organizations that are frequently updated.

The increased complexity of software systems often negatively affect the system dependability. Many recent huge service outages of cloud services resulted from software malfunctions. In February 2017, Amazon Web Service experienced a four-hours of service outage on the east coast of the US, causing widespread problems for thousands of users. According to the analysis by Cyence, S&P 500 companies lost $150 million due to this failure [163]. Amazon described the cause of the problem as an incorrect input command that removed a large set of servers without intention. In March 2017, Microsoft Azure faced problems in their storage availability and provisioning services in 26 out of the 28 regions of the public cloud [164]. Software error and power loss in a storage cluster was announced as the cause of service unavailability. Note that these cloud storage services are built on software technology. Any dormant faults in a software program might cause future service outage, leading to huge impacts on businesses and society. To avoid or mitigate such software-related issues, it is important not only to improve the reliability of individual software components

but also to enhance the efficiency of system management and operation. When a system encounters a failure causing service unavailability, quick recovery from the failure is required, which can be achieved by efficient recovery operation.

In some situations, the occurrence of a failure event can be predicted by the patterns derived from previous experience. Preventive or predictive maintenance could also be effective to improve system availability. Designing and optimizing system management and operation are essential for providing highly available services.

This thesis describes the technologies for providing dependable IT systems and services that are highly available and guaranteed performance even under erroneous conditions by improving system design and operations. To this end, it is important to accurately recognize the design and operation of the target system, discover a potential bottleneck in terms of system availability or performance, and revise the design and operation so that the identified bottleneck is mitigated. System design and operation can be improved continuously by carrying out the process in the plan-do-check-act cycle. Since system availability and performance are affected by various uncertainties including system failure and workload surge, a fundamental challenge is to understand the details of uncertainties and integrate the obtained insight into system design and operation.

1.2. Contributions

In this thesis, several stochastic models are introduced to capture the uncertainties that affect system dependability, performance, and performability (a combined measure explained in Chapter 3). System configuration, maintenance operations, and management policies are modeled with stochastic models so that they can accurately reflect uncertain system dynamics, e.g., component failures, demand arrivals, workload changes, and performance degradations. Various analysis techniques for stochastic models enable the computation of the expected value of measures of interest such as system reliability, availability, performance, and associated costs. The computation can be done on a desktop machine without carrying out expensive real experiments. Therefore, what-if analysis, in which the impact of configuration changes is identified, and sensitivity analysis, where the impact of parameter values is

investigated, can be easily carried out. These analyses techniques are extremely useful to explore the potential improvements of design and operation of complex IT systems. Furthermore, since models can deal with multiple objectives that may conflict with each other, the optimal solution that can satisfy the multiple objectives can be obtained through optimal solution techniques.

The IT systems addressed in this thesis are largely divided into three categories; 1) application service software, which runs continuously and provides service to users, 2) storage systems, which consist of arrays of disks and uses data backup operations, and 3) cloud computing, which uses server virtualization to run operating systems for smart devices. The summary of this thesis' contributions is given below.

- Long-running application-software process: Service systems consisting of continuously running software components often experience performance degradation after long-time execution due to the accumulation of errors caused by software faults. Such a phenomenon is called software aging [26], which could be a precursor to system failure. To prevent system failure after software aging, software rejuvenation is known as a practical countermeasure that clears the accumulated errors by restarting the software-execution environment [44]. Since software rejuvenation stops the execution of the application, which will incur some downtime costs of the system, the schedule to perform rejuvenation needs to be carefully determined in consideration of the dynamics of application services. The analytic model presented in Section 4.2 enables to capture the system behavior with uncertainties and to derive an optimal policy to determine the rejuvenation action depending on the aging status and the expected costs. The derived optimal policy gives a simple guideline for deciding the condition to perform rejuvenation. As a novel countermeasure against software aging that can address the drawback of software rejuvenation, Section 4.3 introduces the technique of software life-extension. The feasibility of this new approach is experimentally evaluated through memcached on a virtual machine. Further analytical study shows that the necessary condition under which exists a unique finite time interval to trigger software life-extension that maximizes the expected system availability. In addition, the effectiveness

of the hybrid approach that combines software rejuvenation with life-extension is presented through numerical studies on proposed stochastic models.

- Storage array and data backup operation: To guarantee the data availability at the base of IT systems, redundant configuration of disk arrays plays an important role in data protection against disk failures. Although there are different choices of disk array configurations, achievable data availability and I/O performance may change depending on the array configurations. Section 5.2 provides a comprehensive stochastic model to quantify the performability of redundant array of independent disks (RAID) that could be useful to select appropriate configuration in consideration with the expected availability and performance. Section 5.2 also shows real benchmark results of common configurations of RAID storage systems that are then supplied to reward values for the presented models to evaluate the performability. The issue of optimal data backup schedule is discussed in Section 5.3. Data backup is necessary to recover data when they are lost due to erroneous operation or storage failure. However, execution of backup consumes resources in a system and can result in the downtime of a service. Considering such requirements, limitations, and costs of backup operation, the backup schedule needs to be designed carefully. Section 5.3 presents a Markov decision process (MDP) approach to formulate the backup scheduling problem and to derive the optimal schedule that satisfies all the requirements for data protection while minimizing total downtime.

- In cloud computing, user applications are hosted on a common computing infrastructure consisting of clusters of servers and virtual machines. Resource management is one of the dependability issues with cloud computing systems, which can affect user-perceived performance as well as service unavailability. To avoid resource contention due to insufficient resources for user demands, effective resource management in the operation of cloud computing is essential. Considering a scenario of a private cloud system for a mobile thin-client service, in which the number of users increases over time and their workloads change day by day, a framework for server procurement decision is presented in Section 6.1. In this framework, stochastic models are used to capture the dynamics of

request arrivals, workload changes, and cloud resource management. Simulation experiments on the proposed models and the real trace data show that the effectiveness of model-based procurement decision over the conventional heuristic approaches.

1.3. Organization

The rest of this thesis is organized as follows. Chapter 2 introduces the metrics of interest considered in this thesis; dependability and performability. Dependability includes various aspects of system reliability following the commonly accepted definition. Performability is regarded as the combined measure of performance and availability, which has become more important in recent highly available systems. Chapter 3 describes some fundamental theories of stochastic models that are used in the subsequent chapters. State-space models such as Markov chains, semi-Markov processes and Markov regenerative processes are briefly reviewed. The introduction to Markov decision process and optimal stopping problem are also provided. Chapter 4 discusses about software aging, rejuvenation, and life-extension. To derive the optimal timing for applying software rejuvenation or life-extension, stochastic models are used to represent the system state transitions. Chapter 5 deals with storage system configuration and data backup operation. Performability model for common RAID storage configuration is presented so as to quantitatively compare the configuration options of storage systems. In addition, optimal data backup schedules are considered with stochastic models which can capture storage failure and available resources. Section 6 introduces a framework to guide server procurement decision in a private cloud system for avoiding performance problems of a mobile thin-client service. This framework is based on simulation on the stochastic models representing request arrivals, workload changes, and resource reallocation processes. Section 7 gives a final remarks and future avenues of research.

Chapter 2

Dependability and performability

This chapter describes the metrics for evaluating the dependability of IT systems. Metrics are important to understand the capabilities of systems in a quantitative and objective manner. Through metrics, system engineers can be guided to find the directions to improve systems and observe the level of improvement after introducing remedies in their design or operation. *Dependability* is one of the focus of this study, which is the concept that subsumes metrics such as reliability, availability, and safety. The commonly accepted definition of the metrics is reviewed in this chapter. Another important measure discussed in this thesis is *performability*, which is a combined measure of performance and availability. In modern IT systems, states of a system cannot always be clearly divided into up or down state; however, there are many intermediate states between the up and down states, where the performance levels are different. Performability is a more suitable measure to quantify the level of system capability for such systems. Related work on these metrics is also briefly explained.

2.1. Dependability metrics

The commonly accepted concept of dependability is given by Avizienis et al [1]. It is defined as *the ability to deliver service that can justifiably be trusted*. The alternate definition of the dependability of a system is *the ability to avoid service failures that are more frequent and more severe than is acceptable*. Dependability is an integrating concept encompassing the following attributes:

- *availability*: readiness for correct service.
- *reliability*: continuity of correct service.

- *safety*: absence of catastrophic consequences on the users and the environment.
- *integrity*: absence of improper system alterations.
- *maintainability*: ability to undergo modifications and repairs.

Availability and reliability are often used as quantitative measures of system dependability, while safety is considered as a qualitative measure. Security is another important concept related to dependability and is regarded as a composite of the attributes of confidentiality, integrity and availability [1]. To attain the various attributes of dependability and security, many means have been developed. Those means are categorized into four major categories:

- *Fault prevention*: means to prevent the occurrence or introduction of faults.
- *Fault tolerance*: means to avoid service failures in the presence of faults.
- *Fault removal*: means to reduce the number and severity of faults.
- *Fault forecasting*: means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention can be achieved by development methodologies and good implementation techniques. A common approach to provide fault tolerance is redundant system configuration, in which redundant components are used to build a system so that a failure of a component does not lead to system failure. Fault removal is critical part of software development as software testing is carried out to minimize the number of software faults contained during coding. Fault forecasting is conducted by performing an evaluation of the system behavior with respect to fault occurrence or activation. Recent advances in big data analysis technique can enhance the capabilities of fault forecasting so that system failure can be avoided before the occurrence. The effectiveness of those means should be evaluated through the metrics corresponding to the attributes such as reliability and availability.

In order to quantitatively evaluate the goodness of those attributes, probabilities are often introduced for reliability and availability measures. *Reliability* can be defined as a function of the time t: $R(t)$ $t>0$, representing the probability that the component survives until time t. $R(t)$ is a monotonically decreasing function of t, assuming that the component is working at

t=0 (i.e., $R(0)$=1) and it can eventually fail (i.e., $\lim_{t\to\infty} R(t) = 0$). *Unreliability* is a complementary measure of reliability which is defined as $F(t) = 1 - R(t)$. It is the probability that the component is failed until time t. Since $R(t)$ as a probability is assigned to the event of component failure, event algebra can be applied to compute the reliability of composite system. For instance, the reliability of a system consisting from component 1 and component 2 in series connection can be computed as $R_s(t) = R_1(t) \cdot R_2(t)$ where $R_i(t)$ i={1,2} represents the reliability of component i. When they are connected in parallel, the system reliability is given by $R_s(t) = 1 - [1 - R_1(t)] \cdot [1 - R_2(t)]$. It can be seen that introducing parallel redundancy is an effective means to improve the whole system reliability.

Availability also can be defined as a function of the time t: $A(t)$ t>0, representing the probability that the component is working properly at time t. Availability is typically defined for repairable system where the failed component can be repaired after some time period. For such repairable system, limiting availability can be characterized by the mean time to failure (MTTF) and mean time to recovery (MTTR) for the component as below

$$\lim_{t\to\infty} A(t) = \frac{\text{MTTF}}{\text{MTTF+MTTR}}.$$

Intuitively deducing that longer MTTF and shorter MTTR contribute to higher system availability. In practice, sometimes system availability is empirically computed by the ratio of up time over the total time including down time. Such a measure is usually used in the quality specification of services. For example, a cloud computing service Amazon EC2 offers 99.95% of availability in their terms of service level agreement (SLA) [2]. This means the service could become unavailable at most 21.6 minutes in a month without any prior notice and if the declared level is not achieved the provider compensates a part of service fee (i.e., penalty). The uptime institute defines the data center tier standards which specify the level of high-availability of the data center [3]. According to the tier standards, tier 4 is ranked as a highest class which requires 99.995% availability. In order to achieve this level, it is mentioned that 2N+1 fully redundant infrastructure is required [4].

In fact, redundant configuration to enhance the reliability of a system is not enough to achieve higher availability. In terms of availability, maintainability also needs to be taken

into the consideration. *Maintainability* addresses the capability of how the system can quickly recover from unavailable condition. Even though MTTF can be prolonged by redundant configuration, lower maintainability could inhibit improving the entire system availability. Similar to the reliability, maintainability sometimes is represented by the probability $G(t)$ $t>0$, indicating the probability that the component recovers until time t. In practice, in the context of business continuity, maintainability of the system is specified as *Recovery Time Objective* (*RTO*) that defines the allowable maximum duration of recovery process. Solution provider offering disaster recovery system often specify the RTO in their service specification. In order to achieve shorter RTO, mirroring data, which synchronize the data with the backup site, might be necessary so that the service on the backup site can take over the operation of the main site at disaster occasion.

2.2. Survivability

Survivability is an extended concept of maintainability and it has been attracting more attention in recent network systems. Survivability is defined as the system's ability to continuously deliver services in compliance with the given requirements in the presence of failures and other undesired events [5]. Based on the definition given by the ANSI T1A1.2 committee, network survivability is quantified as the transient network performance or availability metric from the instant an undesirable event occurs until the steady state where an acceptable metric level is attained [6]. Undesirable events include incidents by malicious attacks, security intrusions and operational ignorance and incompetence leading to system failure. Since recovery process of complex system often span multi-stages, such a transient metric is useful to know the performance impact of the service under recovery [7].

2.3. Resiliency

Resiliency is another important property which recently discussed in dependability research community as well as system industries. Similar to survivability, resiliency focuses on the transient process to recovery the desirable system condition from any changes in the system. The definition of resiliency is provided as "*the persistence of service delivery that*

can justifiably be trusted, when facing changes" by Laurie [8]. Changes impacting on the system performance include functional, environmental, or technical changes in hardware and software. Resiliency is evaluated through the transient behavior of performance measure (e.g., blocking probability, reliability, etc.) during the recovery process [9] or the total amount of deviation from when change occurs to when the desirable performance level is recovered [10][11]. A quantity of resiliency can be formulated as the amount of deviation from the baseline as

$$AD = \int_{t=0}^{T} |p(t) - p_b| dt$$

where $p(t)$ is the performance at time t, p_b is the baseline, and T is the first time when $p(t)$ becomes equal to p_b after change occurs at t=0. The absolute value indicates that the value of $p(t)$ can deviate from p_b to either higher or lower. Regardless of the positive or negative deviations, the measure quantifies the total amount of differences between the observed and target performance values during the recovery from the change at t=0 (see Figure 2.1). Note that the smaller value of AD indicates higher system resiliency.

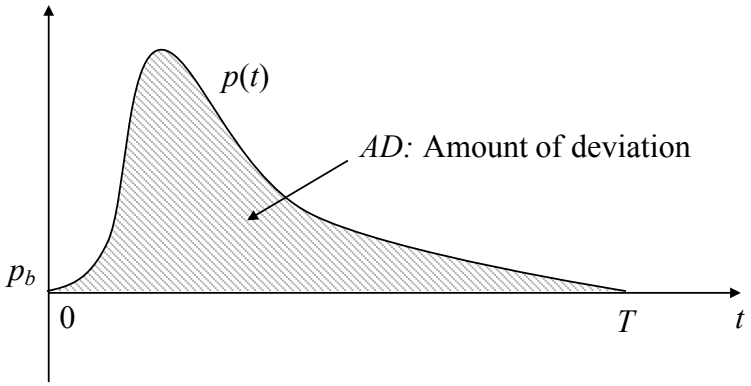


Figure 2.1 Amount of deviation as a quantitative resiliency measure

Amount of deviation as a resiliency quantification has been applied to server virtualized system [10] and video surveillance system [11]. For a server virtualized system, the deviation of availability is evaluated using the model for server virtualized system with rejuvenation [12]. For video surveillance system, by combining system dynamics model with queueing

model, crime risk due the unavailability of surveillance function is considered as a performance measure [11].

2.4. Performability

Performability, a combined terminology of performance and availability by a simple definition, has become an important dependability metric for highly-available systems. As high-availability of critical systems is regarded as a matter of course, the total or average system performance during every operational mode in up times is actual user concern. System needs to be in good performance as long as possible during its uptime. It is desirable that the system can survive with a degraded performance even when a component of system is unavailable. Such a capability cannot be represented by availability nor instantaneous performance.

The notion of performability was originally introduced by J. F. Meyer [13] where the motivation is to evaluate the performance of degradable computing systems. In a degradable computing system, the performance of the system might be degraded depending on the internal state, environment or transitional system configuration. Pure performance evaluation does not generally deal with structural changes of the system, while pure reliability evaluation only takes care of the probability of system failure. A unified performance-reliability measure is required to recognize the effectiveness of such degradable systems. For qualifying the performability, several approaches have been presented. Huslende considered performance reliability by assuming a minimum performance threshold and presented a threshold-based performability measure [14]. Smith et al. evaluated the performability of multiprocessor system by complementary distribution of time-averaged accumulated performance measure [15]. Logothetis et al. [16] presented a general approach to quantify the performability by Markov models with reward assignment. Let $\pi_i(t)$ be the probability of a system being in the state i at time t and p_i is the level of the system performance (e.g., latency, throughput, etc) in state i, the performability can be computed by the product sum of these factors

$$\text{Performability}(t) = \sum_i \pi_i(t) \cdot p_i.$$

Limiting performability can also be defined by $\lim_{t\to\infty} \text{Performability}(t)$. A typical example of degradable system is Redundant Array of Independent Disks (RAID) storage system where the system can tolerant a certain number of disk failures. While a storage system is available with a presence of a single disk failure in RAID5 configuration, the I/O performance must see degraded performance. Figure 2.2 shows a state transition model for RAID5 storage system where each state is labeled with a number of failed disks.

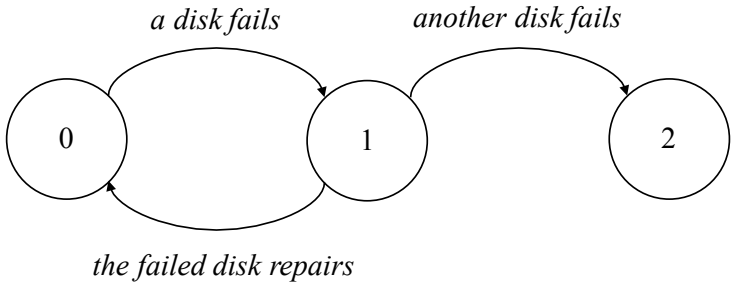


Figure 2.2 State transition model for a RAID5 storage system

The system will fail when two disks are failed before repairing the disk failure at state 1. Consider p_i be the sequential read performance of the storage system when i={0,1} disks fail. The performability of the RAID5 storage system in terms of the sequential read performance can be formulated as $P_{RAID5}(t) = \pi_0(t) \cdot p_0 + \pi_1(t) \cdot p_1$.

Some other related studies about the performability are summarized in [17][18].

Chapter 3

Stochastic models

This chapter introduces the foundations of stochastic models that are used to capture IT system behaviors by taking into account uncertainties to estimate the system performance, availability, and performability. First, general classes of modeling methodologies used in reliability and availability evaluations are reviewed. Then, the basics of state-space models that are used in the later chapters are explained.

3.1. Model classes

There are generally three classes of modeling methodologies commonly accepted for analyzing system performance, reliability, and availability [19]. These are *non-state space model*, *state-space model* and their combination referred to as *hybrid model* in this thesis. Non-state space models have been also called as combinatorial models, since they focus on the combination of functional components in a system. These models abstract the functions necessary for system operation and the capabilities of functional fault-tolerance. However, the dynamic aspects of system behavior are not taken into account in this class of models. State-space models can deal with dynamic behaviors of a system including interactions between system components. For example, a system may perform failover so that a service process in a failed component is taken over by a standby component. While non-state space models cannot represent the failover process, state-space models can capture the process by tracking the state transitions of the system. Table 3.1 summarizes the features of these different modeling methodologies.

Table 3.1 Comparisons among non-state space, state-space and hybrid models

	Non-state space model	**State-space model**	**Hybrid model**
Focus	Configurations of components	State transitions of a system	Both configurations and state transitions
Component behavior	Statistically independent	Dependent of each other	Dependent of each other
Cost for modeling	Low	High	Medium
Cost for solution	Low	High	Medium
Scalability	High	Low	Medium
Reusability	High	Low	High
Examples	Fault tree, RBD, Reliability graph	Markov model, Stochastic Petri net	Hierarchical stochastic model, DRBD, DFT

3.1.1. Non-state space models

With the assumption of statistical independence of events from system components, non-state space model is used to analyze the impact of the states of individual components (e.g., up or down) on the entire system. *Fault tree* clarifies the combinations of components failures leading to entire system failure by connecting the component failure events with *AND*, *OR* and *k-out-of-n* voting gates. Similarly, *reliability block diagram (RBD)* is a non-state space model that represents the series and parallel configurations to provide end-to-end system reliability. In RBD, at least one path of working nodes between the terminal nodes is required to have a system in operation. From the engineering perspective, the graphical representations of non-state-pace models are intuitive and easy to specify by system engineers. There are several efficient solution algorithms for non-state space models to compute reliability, to find a bottleneck and minimum cut sets, and so on. Compared to state-space models, which require state-space enumeration, non-state-space models can generally be efficiently analyzed. Since the rules for connecting events or components are simple, non-state space models can be easily applied to large-scale systems. Non-state models can be

transformed into identical formal expressions with logic operations that have commutative, associative, and distributive properties. Using these operations, it is easy to merge sub-models to construct a larger system model. Sub-models can be reused for modeling other systems consisting of the same types of components. All these good properties of non-state-space models come from the fundamental assumption of statistical independence of component behavior.

3.1.2. State space models

In contrast to stand-alone hardware-oriented systems, recent complex software-based IT systems often have inter-dependent component behaviors. To adequately capture such a dynamic system behavior and analyze the metrics of interest, state-space models are necessary. *Markov models*, described in the next section in details, are representative of state-space models that consist of state definitions, transitions, and associated probabilities. In a Markov model, the probability that a system is in the up state is estimated analytically, which can give an estimation of system availability. Comprehensive state transition involving multiple system components can be taken into account in state-space models. While state-space enumeration works fine with few involved system components, the solution costs of state-space models grow exponentially with the size of the system. Therefore, the scalability of state-space models is considerably limited compared with non-state space models. From the engineering perspective, specifying the state transitions of a system requires significant effort and domain knowledge. Manual enumeration of state spaces is error-prone and time-consuming. Although the graphical representation of state transitions is relatively easy to follow, it becomes notoriously complex when the state-space becomes large. Unlike the non-state-space models, state-space models are not easily reusable since the state transitions of the entire system cannot be divided into independent state transitions of system components.

3.1.3. Hybrid model

Hybrid models are the combination of non-state space model and state-space model to complement the drawbacks of the individual models. A representative example of hybrid

model is *hierarchical stochastic model*, which is composed of non-state space or state-space models in a hierarchical manner for analyzing specific objectives. For example, if the objective of analysis is to compute the entire system availability, a top-level fault tree can represent the system configuration, and a second-level state-space models capture the detailed transitions of individual components. The availabilities of system components are computed individually by solving the state-space models, and their results are supplied to the top-level fault tree so that system-level availability can be computed using a standard fault tree analysis technique. Compared to the approach that solely relies on state-space models, the solution cost is reduced and scalability is further improved. Another hybrid model is created by extending the notations of non-state-space models so that some typical component interactions can be modeled in a non-state-space model's formalism. *Dynamic FT* [20] and *Dynamic RBD* [21] are examples that allow the representation of functional dependency, cold spare, warm spare, priorities, sequential enforcing, and so on. Note that dynamic gate or dynamic blocks introduced in the extended models need to be expanded as state-space models when solving them; thus, the models are essentially regarded as hybrid. As the size of systems in practice increases, hierarchical stochastic models are gaining more attention.

In the remainder of this chapter, as the fundamentals of state-space models, stochastic processes which are used in the later chapters are described.

3.2. Stochastic processes

State-space models are established on the theory of stochastic processes. This section gives a review of the theory of stochastic process including some essentials of probability.

3.2.1. Definition of stochastic process

Probability is assigned to any event from sample space which is an arbitrary set denoted by S. Define a σ-field $\mathcal{F}$ on the sample space S. An event E is an element of $\mathcal{F}$ (e.g., availability monitoring for a system component will see either up or down state). The probability of the event E represents the relative likelihood for the occurrence of E within all

the possible outcomes $\mathcal{F}$. Let $P(E)$ be the probability measure for the event E, the function $P(\cdot)$ satisfies the following Kolmogorov's axioms:

(A1) $\forall E \in \mathcal{F}, P(E) \geq 0$.

(A2) $P(S) = 1$

(A3) For $A_1, A_2, \ldots \in \mathcal{F}$ such that $A_i \cap A_j = \phi\ (i \neq j), P(\cup A_i) = \sum_i P(A_i)$

The expression of event as outcomes of random experiments from $\mathcal{F}$ could be numerical numbers, characters or other symbols depending on the sample spaces considered. In order to abstract such domain-specific representation for mathematical convenience, a real number called *random variable* is used to associate a possible outcome of an experiment. Random variable can be regarded as a measurable function whose domain is $(S, \mathcal{F})$, and whose range is $\left(\mathbb{R}, \mathfrak{B}(\mathbb{R})\right)$ where $\mathbb{R}$ is the set of all real numbers and $\mathfrak{B}(\mathbb{R})$ is a Borel σ-field defined on $\mathbb{R}$.

***Definition* (*Random variable*)**: A random variable X on a probability space $(S, \mathcal{F}, P)$ is a measurable function $X: (S, \mathcal{F}) \rightarrow \left(\mathbb{R}, \mathfrak{B}(\mathbb{R})\right)$ that assigns a real number to each sample point.

The set of all values taken by X is called the image of X. The image of X could be continuous or discrete number. The random variable whose image is continuous number is called *continuous random variable*, while whose image is discrete number is called *discrete random variable*. A random variable X is characterized by a *distribution function* $F_X(x)$ representing the probability that X is equal or less than a real number x,

$$F_X(x) = P(X \leq x), \quad -\infty < x < \infty.$$

The distribution function is also called a *cumulative distribution function (CDF)* of a random variable X. When a CDF is absolutely continuous, the derivative $f_X(x) = dF_X(x)/dx$ is called the *probability density function (pdf)*. The CDF of X can be obtained by integration of pdf:

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} f_X(t)dt, \quad -\infty < x < \infty.$$

The CDF of a random variable X satisfies the following properties:

(F1) $0 \leq F_X(x) \leq 1$.

(F2) $F_X(x)$ is a monotonically increasing function of x.

(F3) $\lim_{x \to -\infty} F_X(x) = 0$ and $\lim_{x \to +\infty} F_X(x) = 1$.

In reliability theory, continuous random variable is often used to associate with the time to failure of system component. In this case, $F_X(x)$ indicates the probability that the system component has been failed at the time x. The exponential distribution is frequently used as an assumed distribution function in reliability model, whose CDF and pdf are given below

$$F_X(x) = \begin{cases} 1 - e^{-\lambda x}, & x > 0, \\ 0, & x \leq 0, \end{cases}$$

$$f_X(x) = \begin{cases} \lambda e^{-\lambda x}, & x > 0, \\ 0, & x \leq 0. \end{cases}$$

where $\lambda > 0$ is the parameter, often called the rate.

An important property of the exponential distribution, so-called memoryless property, is expressed by,

$$P(X > y + x | X > y) = P(X > x).$$

If the random variable X represents the time to component failure, the memoryless property means that the distribution of the residual lifetime does not depend on how long the component has survived so far.

Consider that a value of a random variable corresponds to a certain system state, a family of the random variables can constitute a state-space model. State-space model is based on a stochastic process that is defined as a family of random variables indexed by a certain parameter such as time.

***Definition* (*Stochastic process*)**: A stochastic process is a family of random variables $\{X(t) | t \in T\}$, defined on a given probability space, indexed by the parameter t, where t varies over an index set T.

The set of all the possible values of $X(t)$ is called a state space of the process. The state-space is also called a *chain*, if it is discrete. A stochastic process can describe the state of the system at time t. The state at time t, $X(t)$, is simply a random variable that will follow a certain distribution function. If the conditional distribution of $X(t)$ for given values of all $X(s)$ $s<t$ only depends on the latest value of $X(s)$, the stochastic process $\{X(t)|t \in T\}$ is called a *Markov process* which is formally defined below

***Definition* (*Markov process*)**: A stochastic process $\{X(t)|t \in T\}$ is called a *Markov process* if for any $t_0 < t_1 < \ldots < t_n < t$, the conditional distribution of $X(t)$ for given values of $X(t_0)$, $X(t_1)$, …, $X(t_n)$ depends only on $X(t_n)$:

$$P[X(t) \leq x|X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \ldots, X(t_0) = x_0] = P[X(t) \leq x|X(t_n) = x_n].$$

The Markov process is said to be (time-) homogeneous if the conditional distribution of $X(t)$ also has the property of invariance with respect to the time origin t_n:

$$P[X(t) \leq x|X(t_n) = x_n] = P[X(t - t_n) \leq x|X(0) = x_n].$$

For a homogeneous Markov process, the past history of the process is completely summarized in the current state. Considering a homogeneous Markov chain, which is a discrete-state Markov process, the distribution for the time spent in a specific state does not depend on how long it has already spent in that state. This indicates that the time that a homogeneous Markov chain spends in a given state has memoryless property. In a continuous-time Markov chain, the time spent in a given state follows an exponential distribution which satisfy the memoryless property. If the distribution of the sojourn time in a specific state is not assumed to be an exponential distribution, the stochastic process becomes a semi-Markov process described later.

3.2.2. Poisson process

The *Poisson process* is a continuous-time discrete-state stochastic process that is often used for modeling random arrivals of requests, demands or events to a system. Considering the number of events $N(t)$ occurring in the time interval $(0, t)$, the stochastic process

$\{N(t)|t \geq 0\}$ is called a Poisson process if the time intervals between successive events are independent and identically distributed (*iid*) according to an exponential distribution. In a Poisson process, $N(t)$ in the interval $(0, t)$ is characterized by a Poisson probability mass function (*pmf*) with parameter $\lambda > 0$ as given by

$$p_n(t) = e^{-\lambda t}\frac{(\lambda t)^n}{n!}, \quad n > 0.$$

The parameter λ in the Poisson process is called arrival rate. If the arrival rate changes depending on the time *t*, the Poisson process is called the *non-homogeneous Poisson process (NHPP)*. In an NHPP, the distribution of the number of arrivals $N(t)$ follows Poisson distribution with parameter $m(t) = \int_0^t \lambda(x)dx$ that is called mean-value function. One of the important applications of NHPP has been extensively studied as software reliability growth models [22].

3.2.3. Discrete-Time Markov Chain

The Markov chain whose parameter space *T* is discrete is called a *discrete-time Markov chain (DTMC)*. Denote X_n as a random variable at the time step $n \in T$.

***Definition* (*DTMC*)**: A stochastic process $\{X_n, n \geq 0\}$ with countable state-space *S* is called a DTMC if it has the Markov property

$$P[X_{n+1} = j|X_n = i, X_{n-1}, X_{n-2}, \dots, X_0\,] = P[X_{n+1} = j|X_n = i\,].$$

In a time-homogeneous DTMC, the conditional probability that $X_{n+1}=j$ for given $X_n=i$ does not depend on the value of *n*.

$$P[X_{n+1} = j|X_n = i\,] = p_{i,j}, \quad \forall n \geq 0, \qquad i,j \in S$$

The conditional probability $p_{i,j}$ is called one-step transition probability. The matrix of the one-step transition probabilities $\boldsymbol{P}=[p_{i,j}]$ is called transition probability matrix. It is known that DTMC is completely described by the transition probability matrix $\boldsymbol{P}$ and the initial distribution of X_0. Denote $\boldsymbol{p}(n)$ as the row vector whose *j*-the element is $P(X_n=j)$. $\boldsymbol{p}(n)$ can be described as $\boldsymbol{p}(n)= \boldsymbol{p}(0)\boldsymbol{P}^n$ where $\boldsymbol{p}(0)$ is the initial provability vector.

A graphical representation of DTMC can be given by a state-transition diagram as shown in an example in Figure 3.1. A node labeled *i* represents state *i* of the Markov chain and a label attached to the directed arc from node *i* to node *j* represents the one-step transition probability $p_{i,j}$.

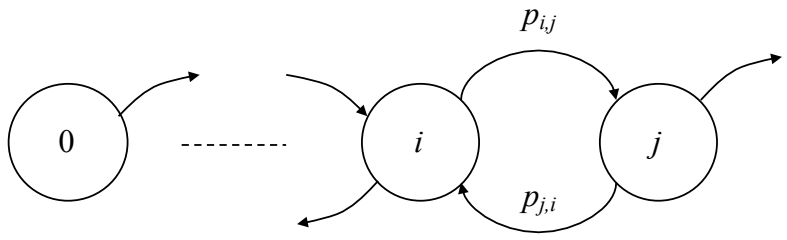


Figure 3.1 An example of a graphical representation of DTMC

A Markov chain is said to be *irreducible* if every state can be reached from every other state in a finite number of steps. For a specific type of DTMC mostly used in this article, the following theorem to characterize the steady-state behavior of DTMC is applicable.

Theorem 3.1: For an irreducible, aperiodic and positive-recurrent Markov chain, the limiting probability vector $v = [v_0, v_1, \ldots]$ is the unique stationary probability vector satisfying

$$v_j = \sum_j v_i p_{i,j}\,, \quad j = 0,1,2, \ldots,$$

$$v_j \geq 0, \qquad \sum_j v_i = 1.$$

v is also known as the steady-state probability vector.

If state-space is finite, with this theorem, *v* can be obtained by solving a system of linear equations.

3.2.4. Continuous-Time Markov Chain

If the parameter space *T* of the Markov chain is continuous, the Markov chain is called a *continuous-time Markov chain (CTMC)*. In a CTMC, the transition from a given state to another state can take place at any instant of time.

***Definition* (*CTMC*)**: A discrete-state continuous-time stochastic process $\{X(t)|t \geq 0\}$ is called a CTMC if it holds the Markov property

$$P[X(t) = x|X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \dots, X(t_0) = x_0\ ] = P[X(t) = x|X(t_n) = x_n\ ]$$

for $t_0 < t_1 < \ldots < t_n < t$.

In a time-homogeneous CTMC, the transition probability from state i to state j depends only on the time difference:

$$p_{i,j}(t) = P[X(t+v) = j|X(v) = i], \qquad \forall v \geq 0$$

Denote $\pi(t)$ as the probability vector whose j-th element is $P(X(t)=j)$. The probabilistic behavior of a CTMC is completely determined by the transition probability matrix $\boldsymbol{P}(t)=[p_{i,j}(t)]$ and the initial probability vector $\pi(0)$ by

$$\pi(t) = \pi(0)\boldsymbol{P}(t)$$

Define the infinitesimal generator matrix $\boldsymbol{Q}=[q_{i,j}]$ for a time-homogeneous CTMC

$$\boldsymbol{Q} = \lim_{t \downarrow 0} \frac{\boldsymbol{P}(t) - \boldsymbol{I}}{t}.$$

The probability vector and the infinitesimal generator matrix have the following relationship.

$$\frac{d\pi(t)}{dt} = \pi(t)\boldsymbol{Q}$$

For an irreducible CTMC with finite state-space, the limiting probabilities $\pi_j = \lim_{t \to \infty} \pi_j(t)$ always exist. The steady-state probabilities can be computed by the following theorem.

Theorem 3.2: For an irreducible regular Markov chain with finite state-space, the limiting probability vector $\boldsymbol{\pi} = [\pi_0, \pi_1, \dots]$ is the unique stationary probability vector satisfying

$$\boldsymbol{\pi}\boldsymbol{Q} = 0,$$

$$\sum_j \pi_i = 1.$$

$\boldsymbol{\pi}$ is also known as the steady-state probability vector.

The unique limiting probabilities $\boldsymbol{\pi}$ can be obtained by solving a system of linear equations $\boldsymbol{\pi Q} = 0$ under the condition $\sum_j \pi_i = 1$.

3.2.5. Semi-Markov process

Semi-Markov process (SMP) is a generalization of CTMC by relaxing the exponential sojourn time assumption [23]. CTMC requires that the stochastic process satisfies Markov property at every time *t* that restricts the transition time distributions to be exponential. In contrast, SMP requires Markov property only at every jump epoch and hence allows general distributions for state transition times. Consider the process stays in state X_n, $n \geq 0$ for a duration given by Y_{n+1} and then jumps to state X_{n+1}. The sequence $\{X_0, (X_n, Y_n), n \geq 1\}$ is characterized by a stochastic process $X(t) = X_{N(t)}, t \geq 0$, where $N(t)$ is the number of jumps up to time *t*. The definition of SMP can be given by below.

***Definition* (*Semi-Markov Process*)**: The stochastic process $\{X(t)|t \geq 0\}$ is called a homogeneous semi-Markov Process (SMP) if $X(t) = X_{N(t)}$ where $N(t)$ represents the number of jumps up to time *t* and the sequence $\{X_0, (X_n, Y_n), n \geq 1\}$ satisfies

$$P[X_{n+1} = j, Y_{n+1} \leq y | X_n = i, Y_n, X_{n-1}, Y_{n-1}, \dots, X_1, Y_1, X_0\,] = P[X_1 = j, Y_1 \leq y | X_0 = i\,]$$

for $i, j \in S, n \geq 0$.

The conditional transition probabilities $K_{i,j}(y) = P[X_1 = j, Y_1 \leq y | X_0 = i\,]$, $i, j \in S$ is called the *kernel* of an SMP. Taking the limit of the kernel $p_{i,j} = K_{i,j}(\infty)$ yields the transition probability matrix of $\{X_n, n \geq 0\}$ that is a DTMC also called an *embedded Markov chain*. Meanwhile, the summation of the kernel distribution from the same state *i*

$$h_i(t) = \sum_j K_{i,j}\,(t)$$

gives the sojourn time distribution in state *i*. With this transition probability matrix $\boldsymbol{P}$ and the sojourn time distributions $\boldsymbol{H}(t) = (h_1(t), h_2(t), \dots)$, steady-state solution of an SMP can be derived by the two-stage method. The mean sojourn time in state *i* is given by

$$h_i = \int_0^{\infty} [1 - h_i(t)]\, dt.$$

In the first stage, steady-state probability vector of embedded Markov chain is computed by ***v*=*vP***. In the second stage, using the mean sojourn time, the steady-state probability vector of the SMP can be derived as

$$\pi_i = \frac{v_i h_i}{\sum_{k=0}^{n} v_k h_k}$$

3.2.6. Markov regenerative process

A stochastic process $\{Z(t)|t \geq 0\}$ with state space *S* is called *regenerative* if there exist time points at which the process probabilistically restarts itself. Such random time points when the future of the process becomes a probabilistic replica of itself are named *regeneration points*. In an SMP, all the state transition time points are categorized into regeneration points as the transition probabilities only depend on X_0. In other words, all the state transients in SMP are associated with regeneration points. If a regenerative stochastic process allows non-Markov state transitions between regeneration points, the process is no more SMP and is called a *Markov regenerative process (MRGP)* [23]. The definition of MRGP is given as below

***Definition* (*Markov regenerative process*)**: A stochastic process $\{Z(t)|t \geq 0\}$ with state-space *S* is called a Markov regenerative process (MRGP) if there exists a Markov renewal sequence $\{(X_n, S_n), X_n \in S, n \geq 0\}$, such that all the conditional finite distributions of $\{Z(t + S_n)|t \geq 0\}$ for given $\{Z(u): 0 \leq u < S_n, X_n = i\}$ are the same as those of $\{Z(t)|t \geq 0\}$ for given X_0=*i*.

The definition implies that

$$P[Z(t + S_n) = j|Z(u): 0 \leq u < S_n, X_n = i\,] = P[Z(t) = j|X_0 = i\,].$$

The Markov property holds only on the sequence of time points (S_0, S_1, …) such that the states (X_0, X_1, …) respectively of the process at these points. The future of the MRGP from $t=S_n$ depends on its past up to time S_n only through X_n.

Similar to SMP, the kernel of MRGP can be given by the conditional transition probabilities

$$K_{i,j}(t) = P(X_{n+1} = j,\ S_{n+1} - S_n \leq t | X_n = i)$$

The matrix of kernel distribution $\boldsymbol{K}(t)=[K_{i,j}(t)]$ is called the *global kernel* of the MRGP. The global kernel capture the stochastic behavior of embedded Markov chain of MRGP. On the other hand, the behavior between two consecutive regeneration points is captured by the probabilities that the process in state j at time t starting from the state i before the next regeneration time point

$$E_{i,j}(t) = P[Z(t) = j, S_1 > t | Z(0) = i]$$

The matrix $\boldsymbol{E}(t)=[E_{i,j}(t)]$ is called the *local kernel* of MRGP.

When the embedded discrete time Markov chain is finite and irreducible, its steady-state probability vector v is given by the solution of the linear system $v = v\boldsymbol{K}(\infty)$ under the condition $\sum_i v_i = 1$. Then the steady-state probabilities π_j are given by

$$\pi_j = \frac{\sum_{k\in\Omega} v_k \alpha_{kj}}{\sum_{k\in\Omega} v_k \sum_{l\in\Omega} \alpha_{kl}}$$

where $\alpha_{ij} = \int_0^\infty E_{ij}(t)dt$ is the mean sojourn time in state j before the next regeneration point, given the initial state i.

3.3. Decision processes

Stochastic processes described above sections model the stochastic behavior of systems given all the transitions in the state space S. When there exists options a decision maker controls state transition via a certain action, the process is extended to a *decision process*. In a decision process, a set of actions which a decision maker can take and conditional transition probabilities for given action are specified in addition to the state space. Decision of action

is made for a certain objective which can be modeled by reward or cost assignment to possible actions for given state. A particular class of decision process studies in this article is *Markov decision process (MDP)* [25] which extends a Markov process. The definition of MDP is given in the following.

3.3.1. Markov decision process

***Definition* (*Markov decision process*)**: A stochastic control process with state space *S* and a set of actions *A* is called a Markov decision process if the conditional transition probabilities for given state $x \in S$ and action $a \in A$ depend on the previous states and actions only through the current state *x* and action *a*. The definition implies that:

$$P[X(t) = x | X(t_n) = x_n, Y(t_n) = a_n, X(t_{n-1}) = x_{n-1}, Y(t_{n-1}) = a_{n-1}, \dots] = P[X(t) = x | X(t_n) = x_n, Y(t_n) = a_n]$$

where $X(t)$ and $Y(t)$ are the state and action at time t, respectively, for $t_0 < t_1 < \ldots < t_n < t$.

Define reward function $r(x, a)$ which assign the expected reward for action a in state x. The solution of MDP is given by the optimal policy that determines the optimal decision for every state in *S* in terms of expected reward. For an MDP with infinite time horizon, the optimal policy can be derived by solving the *optimal Bellman equation* [24] given by

$$V^*(x) = \max_{a \in A} \left[r(x, a) + \gamma \sum_y p(y|x, a) \cdot V^*(y) \right],$$

where γ is a discount factor, $0 \leq \gamma < 1$, representing the relative importance of recent values. The optimal policy is given by

$$\pi^*(x) = \arg\max_{a \in A} \left[r(x, a) + \gamma \sum_y p(y|x, a) \cdot V^*(y) \right].$$

To obtain the solution of the optimal Bellman equation which is a non-linear system of equations, dynamic programming is often employed. Two typical dynamic programming

techniques are known as *value iteration* and *policy iteration*. Value iteration starts from a guess value of V_0 and compute

$$V_{k+1}(x) = \max_{a \in A} \left[R(x,a) + \gamma \sum_{y} p(y|x,a) \cdot V_k(y) \right]$$

at each iteration k=1, 2, …, K, and return the policy

$$\pi_K(x) = \arg\max_{a \in A} \left[R(x,a) + \gamma \sum_{y} p(y|x,a) \cdot V_K(y) \right].$$

Each iteration process is computationally efficient, but it may require a large number of iterations for satisfactory result. On the other hand, policy iteration starts from any stationary policy π_0 and at each iteration k=1, 2, …, K, derives V^{π_k} and computes the improved policy

$$\pi_{k+1}(x) = \arg\max_{a \in A} \left[R(x,a) + \gamma \sum_{y} p(y|x,a) \cdot V^{\pi_k}(y) \right].$$

Although each iteration is computationally expensive, the number of iterations tends to be small.

3.3.2. Optimal stopping

Consider a special class of decision process includes an action to terminate the process itself. At every decision point, a decision maker can take either stop action a_s or wait action a_w. When the stop action is selected, the process terminates immediately and receives an associated reward or cost. On the other hand, when the wait action is selected, the process continues until the next decision point and associated reward or cost is given. The total expected reward or cost at the end of the process is often the criteria to determine the optimal state to take the stop action. The problem to find the optimal policy for the decision process with a stop action is called an *optimal stopping problem* [25]. The optimal policy can be derived by the following optimal Bellman equation

$$V^*(i) = \max\left\{r_i, \gamma_i \sum_j p_{i,j} \cdot V^*(j)\right\},$$

where r_i and γ_i are the rewards gained by stop action and wait action in state i, respectively and $p_{i,j}$ is the transition probability from state i to state j. The solution of the optimal Bellman equation can be obtained through applying dynamic programming.

Chapter 4

Software aging, rejuvenation and life-extension

As IT systems are becoming more software-intensive than ever, the reliability and availability of such systems are also highly sensitive to the quality of software components. Most software malfunctions are caused by dormant software faults introduced in the coding that are not detected in the testing stages. Therefore, dealing with such software faults, so-called software bugs, over the system lifecycle is a clue to improve the system availability. In this chapter, the impact of a specific type of software fault on the availability and performance of IT systems are discussed. In particular, *aging-related software bug* whose problem can manifest only after long continuous execution is focused. Using stochastic models, the effectiveness of countermeasures against software aging is analyzed. The chapter is organized as follows. First, the definition and examples of software aging are explained in Section 4.1. There are two known countermeasures against software aging problem; software rejuvenation and life-extension. Section 4.2 focuses on software rejuvenation methods and provide a comprehensive review of the concept and modeling techniques for software rejuvenation. Subsequently, our original study that derives the optimal timing to trigger software rejuvenation in terms of job processing performance is explained. We formulate the optimal rejuvenation scheduling problem as an optimal stopping problem and give its solution. Section 4.3 proposes software life-extension which is a new approach to mitigate the impact of software aging. The effectiveness of the approach is presented through the modeling and analysis of SMP. Some experimental results using a virtual machine for software life-extension are also presented.

4.1. Software aging

4.1.1. Definition

Software aging is a phenomenon of progressive degradation of software execution environment caused by software faults. The manifestation of software aging usually occurs after a long time continuous execution of the software and it leads to adverse consequences such as serious performance degradation and/or system failure. A typical example of software aging is a progressive increase in memory consumption that eventually causes a memory leak failure. The software faults inducing such software aging problem are called aging-related bugs [26][27]. Such a type of bugs is found in a wide variety of software products such as space mission system [28], cloud computing software libraries [29], Linux OSes [30], and Android mobile OSes [31].

The consequences of software aging are commonly observed as software hung in computing devices mostly caused by memory leak. Sometimes such failures have a huge impact on our expensive mission or safe lives. It was reported that Mars Exploration Rovers launched by NASA contains the software bug leading to memory exhaustion in the FLASH that inhibits the exploration mission [32][33]. A significant incident caused by software aging is the out of control of Patriot missile-defense system that leads to 28 US Army reservists' deaths and 97 injured [26]. The system failed to track the target missile due to the imprecisions of the time values arose after long runtime without reboot. The Army officials assumed that users would not continuously operate the Patriot systems long enough for a failure to become imminent.

According to [31], there are five common types of aging-related bugs.

- MEM: Aging-related bugs causing the accumulation of errors related to memory management (e.g., memory leaks, buffers not being flushed);
- STO: Aging-related bugs causing the accumulation of errors that affect storage space (e.g., the bug consumes disk space);

- LOG: Aging-related bugs causing leaks of other logical resources, that is, system-dependent data structures (e.g., sockets or i-nodes that are not freed after usage);
- NUM: Aging-related bugs causing the accumulation of numerical errors (e.g., round-off errors, integer overflows);
- TOT: Aging-related bugs in which the increase of the fault activation/error propagation rate with the total system run time is not caused by the accumulation of internal error states.

From the empirical investigations from bug reports for Linux, MySQL, Apache HTTPD and Apache AXIS, it was reported that MEM was the major sources of software aging.

4.1.2. Examples

Real examples of software aging observed in the system using popular software products are described below.

Software aging observed in UNIX workstations was statistically investigated by Garg et al. in early study [34]. Simple Network Management Protocol (SNMP) is adopted for monitoring the resource usages and system activity data at regular time intervals from networked UNIX workstations. The trend in the collected data is detected by seasonal Mann-Kendall test and the slope is estimated by Sen's non-parametric procedure. During the 53 days of the experimental period, UNIX workstations actually experienced the outages due to software aging in real memory or swap space. Using statistical techniques, Garg et al. detected the trends in resource usages and computed the estimated time to reach exhaustion.

Using similar statistical approach, Cotroneo et al. analyzed the software aging phenomenon in Linux Operating system [30]. An actual software aging in memory consumption of Linux OS was observed as shown in Figure 4.1. In order to delve into the root-cause of software aging, a special kernel tracing tool was developed for collecting the data from kernel activities. As a result of the collected data analysis, it was found that the caching mechanism for filesystem data structure caused the increasing trend of memory consumption.

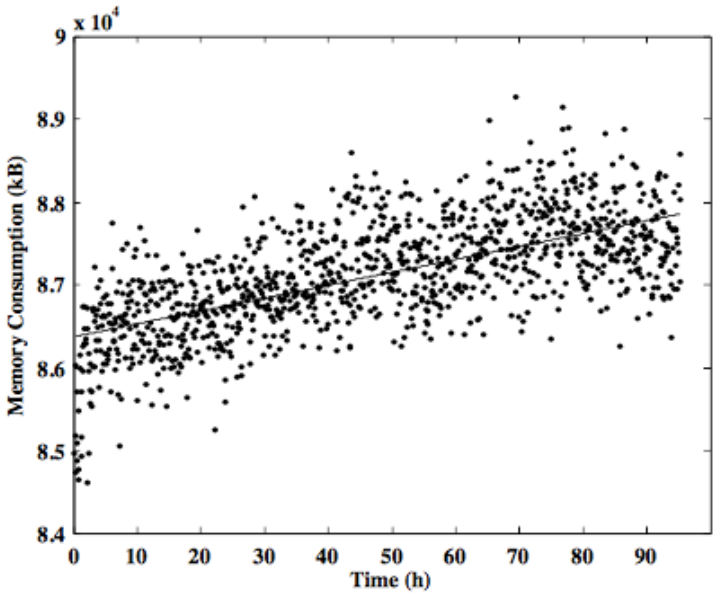


Figure 4.1 Increasing trend in memory consumption of Linux OS [30]

Machida et al. reported the real software aging phenomenon observed in Xen's virtualized system [35]. Two different types of software aging in Xen's hypervisor are uncovered by the stress tests. A software aging problem is observed in the system using Xen 3.0 in which the size of free memory gradually decreases according to the number of VM migration operations (See Figure 4.2).

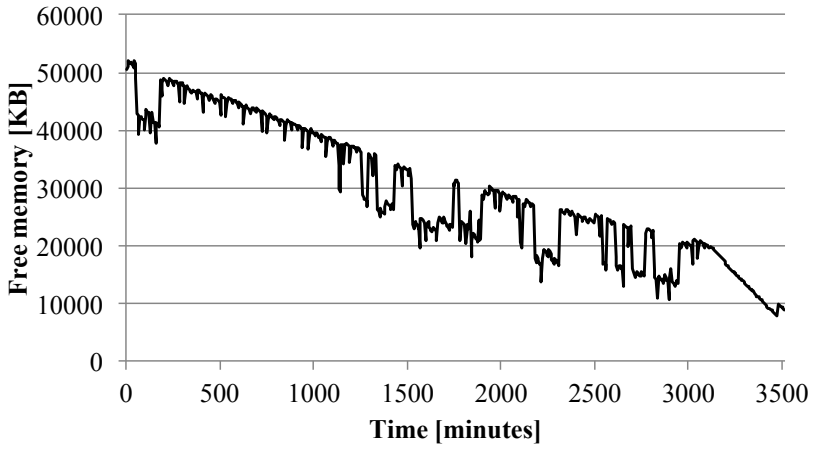


Figure 4.2 Software aging in free memory on Xen 3.0 hypervisor [35]

The other software aging occurs in free disk space in the system installing Xen 3.1. It is reported that 185MB temporal file is created each time when VM suspend operation. Due to the software bug, the temporal file is not removed even after VM resume, resulting in unnecessary occupation of free disk spaces (See Figure 4.3).

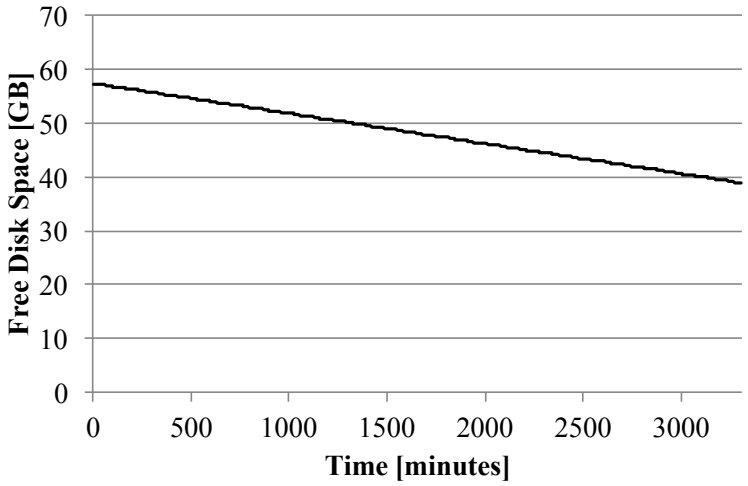


Figure 4.3 Software aging in free disk space on Xen 3.1 hypervisor [35]

While the majority of software aging is associated with memory management problem, Cotroneo et al. spotlighted the numerical software aging issues [36]. Through the investigation of bug reports of MySQL database management system (DBMS), many potential integer overflow problems are discovered. For example, long runtime of DBMS instance might cause a failure of `count()` function due to the limit size of tuples in a table, which is 2^{32}-1. This issue can be considered a software aging problem, since the likelihood of an integer overflow increases with the amount of insert operations that have been performed. Another example of integer overflow can be seen in the status variables in a DBMS server process. The values of some status variables increase with the amount of operations processed by the DBMS, leading to an overflow after some days. The percentage of full table scans, which is the percentage of operations that require to sequentially inspect the tuples of a table, is affected by the overflowed values of *Handler_read_rnd_next* and *Handler_read_rnd*. It is mentioned that these variables can overflow after few days of execution in a highly-loaded DBMS instance.

Recent study by Cotroneo pointed out that mobile OSes also faced the risk of a software aging leading to gradual degradation of user-perceived performance [31]. The experimental studies are conducted for Android mobile OSes to characterize software aging phenomena. After few hours of stress testing, a noticeable performance degradation in terms of low responsiveness of the system has been observed. Then the specific processes that cause the performance degradation by consuming an increased amount of memory were identified. The processes, system server, surface flinger, and system UI, have the largest impacts on the increase in memory consumption. It is recommended that measurement-based software rejuvenation should be adopted to clear the software aging issue before its impact become immense.

More real examples can be found in the literature [37].

4.1.3. Mitigations

Mitigation of software aging impacts can be achieved through some fault-tolerant, removal or avoidance techniques for software components. In software development phase, software testing is a key to detect and remove aging-related bugs as well as other common types of bugs. Compared to other types of bugs, aging-related bugs tend to remain even after some functional tests, since whose activation highly depends on its execution environment (e.g., the amount of resources and workloads). Therefore, systematic tests using a test execution environment are important to find aging problem in the development phase. In software operational phase, on the other hand, it is not easy to locate the aging-related bugs even when experiencing software aging. Instead of finding the root-cause, operational countermeasures can provide an effective solution to software aging mitigation. Software rejuvenation is known as a simple but very effective countermeasure which can avoid preventively a serious consequence of software aging. Software life-extension is another effective approach to mitigate the impacts of software aging without interrupting the continuous execution of application services. Related works for software testing, rejuvenation and life-extension is described in the following sub-sections, respectively.

4.1.3.1. Software testing

Detection of software aging in the development phase is challenging, because it takes a long time to observe software aging, and one also faces problematic false alarms. Accelerated life-time test [38] and degradation test [39] are capable of reducing the time needed to observe software aging phenomena. While accelerated lifetime test was originally developed for hardware lifetime test, Matias et al applies the technique for lifetime analysis of software suffering from aging [38]. The lifetime distribution is estimated from the real lifetime data obtained through stress test with different stress levels. Accelerated degradation test is an extension of accelerated lifetime test and is applied for highly reliable system which can have long lifetime. It focuses on estimating the degradation measure instead of the time to failure. Software aging phenomena can be characterized by analyzing the relationship between degradation and stress levels used in the experiments [39]. In terms of false alarms, common statistical test approach is not enough to distinguish software aging from benign trends in resource usage [40]. Matias et al. introduced divergence charts to characterize abnormal trends in resource usage collected by controlled experiments [41]. Selecting a relevant aging indicator to be monitored and analysis with divergence chart are keys to faster detection of software aging by tests.

Finding the root cause of software aging is further difficult even if software aging is detected by experiments. To locate the source of the problem, it is required to delve into suspicious source code in reference to observations of software aging. Since exhaustive search of entire source code is troublesome and expensive task, the way to narrow down the scope of analysis is important to effective and fast debugging. Felix et al. proposed a version comparison approach that compares the results of stress tests in the connective software versions so that test if a new version of source code is contaminated by aging-related bugs [42]. This approach effectively localizes the domain of rood cause in the entire source code when software aging is observed in the execution of a new version of software.

Even with state-of-art testing and debugging techniques, complete removal of aging-related bugs is practically infeasible or unacceptably expensive. Instead of removing the bugs completely in the development phase, it may be more cost-effective to deal with the problem in the operational phase. The trade-off between the efforts to software testing and potential

unavailabilities encountered in operational phase was studied by Grottke et al [43]. In practice, it is important to allocate the budgets to testing and operation in a balanced manner in consideration with required system availability.

4.1.3.2. Software rejuvenation

Software rejuvenation is an operational countermeasure to software aging. It is typically conducted by restarting entire system and clear the accumulated errors caused by aging [44]. Although the applications running on the system needs to be interrupted in a short period of time, the system can restore its full performance and robust execution environment. In stack of software components consisting a system, there are levels of granularities rejuvenation is applied. When smaller granularity of software component is chosen for rejuvenation target, the down time caused by rejuvenation tends to be shorter. For example, software rejuvenation in application level could complete in a shorter time than rejuvenation by restarting OS. Alonso et al. categorized rejuvenation granularities in six categories; (i) node level, (ii) virtualization level, (iii), operating system level, (iv) OS component level, (v) application level, and (vi) application component level [45]. The overheads of rejuvenation in different level were experimentally evaluated in their study.

Software rejuvenation techniques can be classified into two major categories in terms the timing decision of rejuvenation action. Trigger of software rejuvenation could be either time-based or condition-based. The former decided the rejuvenation in a predetermined fixed time interval. No matter what condition the system is in, a rejuvenation is started at a scheduled time. On the other hand, condition-based rejuvenation is performed only when a specific condition of system status is satisfied. Condition-based rejuvenation is usually introduced with system monitoring that monitor the values of specific system metrics online and check whether the value reaches to a predefined threshold. While this approach can effectively reduce the unnecessary rejuvenation triggers, its effectiveness relies on how to select an appropriate threshold, which is not always straightforward due to uncertainties of software aging manifestation. Okamura et al. presented a more practical approach to decide rejuvenation timing called opportunity-based rejuvenation [46][47]. It is essentially a

combination of time-based and condition-based rejuvenation. Rejuvenation is basically scheduled at a fixed time interval. Instead of starting rejuvenation immediately at rejuvenation schedule, the actual trigger of rejuvenation is postponed until a specific condition is satisfied. Such a condition is called *opportunity*, which is determined by some operational constraints in system administration.

4.1.3.3. Software life-extension

As an operational countermeasure to software aging, a variety of software rejuvenation techniques and models have been studied for a long time. However, most the proposed techniques did not reach beyond the original concept of software rejuvenation. As a novel countermeasure, Machida et al. [48] proposed a concept of software life-extension and studies its feasibility and effectiveness. Similar to software rejuvenation, software life-extension is a preventive and operational measure to software aging problem. Instead of stopping the execution as rejuvenation does, the approach attempts to slow down the progress of software aging so as to prolong the lifetime as long as possible. This approach is particularly beneficial for applications require continuous operation for a specified mission period. For example, a simulation application spawns a long running job requires a continuous runtime as its intermediate results during the execution will be cleared at execution interruption (i.e., software rejuvenation is not a solution). There are two conceptual ways to achieve software life-extension that are (i) allocating additional resources to the system, and (ii) reducing the workload which advances software aging. The details of software life-extension are presented in Section 4.3.

4.2. Software rejuvenation

According to a comprehensive survey carried out by Cotroneo et al. [37], studies about software rejuvenation can be broadly classified into two categories; model-based approach and measurement-based approach. Model-based approach is particularly important to assess the effectiveness of software rejuvenation in terms of system availability and performability by design. While a number of studies aimed to maximize the system availability or to

minimize the downtime cost, less work has been conducted in the analysis of rejuvenation impacts on application performance. This thesis mainly focuses on model-based approach and deal with the rejuvenation scheduling problem which aims to minimize the cost in terms of application performance. In this section, first the literature review of model-based studies for software rejuvenation is provided. Then, Section 4.2.2 explains one of our contributions to software rejuvenation research that gives an optimal policy for deciding software rejuvenation which minimizes the performance cost of a job processing system [49]. Section 4.2.3 describes an extended work in which some assumptions introduced in [49] are relaxed and an optimal stopping problem for software rejuvenation decision is reformulated [50]. The analytical solution gives a simple but powerful guideline for determining a trigger of software rejuvenation.

4.2.1. Related work on rejuvenation models

This section is dedicated for the literature survey on model-based approach of software rejuvenation. The part of the content will be appeared in a book chapter of Software Aging and Rejuvenation Handbook [51], in which the content is prepared in collaboration with Prof. Paulo Maciel in Federal University of Pernambuco.

Starting from the original rejuvenation model presented by Huang et al. [44] which was based on Continuous Time Markov Chain, a wide variety of Markov and semi-Markov models have been used for representing different types of rejuvenation policies, techniques, and systems. Semi-Markov model allows to incorporate a deterministic trigger of software rejuvenation in the model. The optimal interval to trigger software rejuvenation can be analyzed by the solution technique for semi-Markov process. Another important evolution of Markov model for software rejuvenation can be seen in the models based on Stochastic Petri Nets and their variants that are suitable for modeling more complex and large-scale systems. The earliest study in this realm was Markov Regenerative Stochastic Petri Nets presented by Garg et al [52]. Further complex systems are analyzed by modeling with Fluid Stochastic Petri nets, Stochastic Reward Nets, Deterministic Stochastic Petri Nets and hierarchical models etc. The history of evolution of these models are also reviewed in this section.

4.2.1.1. Introduction

As mentioned previously, software rejuvenation is known to be an effective countermeasure to software aging by resetting the system prior to encounter system failure. Since the downtime cost incurred by software rejuvenation is smaller than the cost caused by system failure, it is desirable to perform software rejuvenation proactively before encountering serious problems caused by aging. However, it is not a trivial issue to determine when to trigger software rejuvenation because the process of software aging and associated failures cannot be captured as a deterministic process and need to deal with uncertainties. The effectiveness of software rejuvenation relies on various factors of uncertainties such as the time to software aging, the time to failure, the time to recovery, the time to rejuvenation and the associated costs. To deal with such processes, Markov model is a powerful mathematical tool to abstract the dynamics of systems and to analyze the relations of those factors contributing to system availability and performance. This section reviews the modeling approach based on Markov models to characterize the behavior of software aging and rejuvenation systems. The content is largely divided into two parts; the part for Markov chains and semi-Markov processes and the other part for Petri Nets. Figure 4.4 shows a history overview of modeling studies covered in this review.

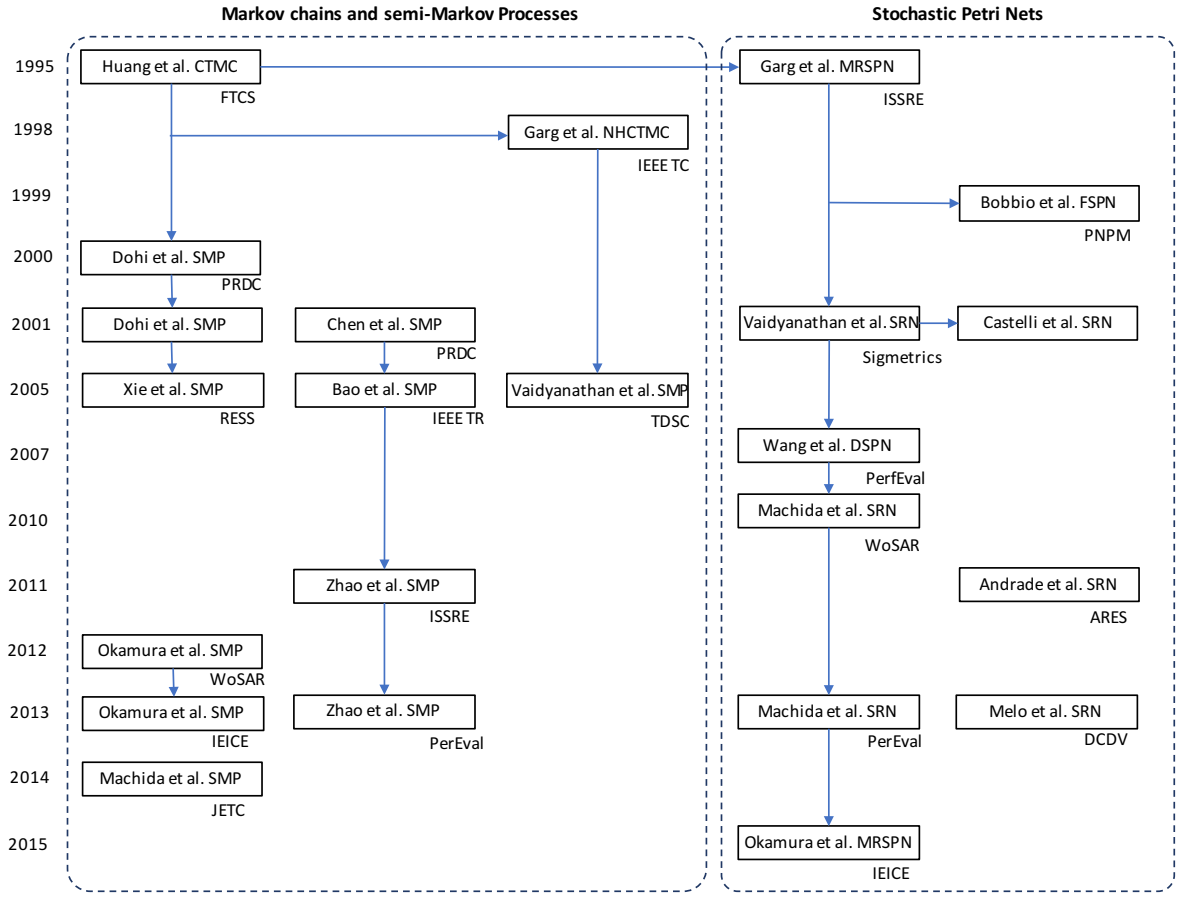


Figure 4.4 A brief history of Markov chains and stochastic Petri nets applied to software aging and rejuvenation [51]

4.2.1.2. Markov chains and Semi-Markov processes

The first application of Markov model to analyze the dynamics of software aging and rejuvenation was presented by Huang, Kintala, Kolettis and Fulton [44]. The authors proposed a Continuous-time Markov Chain (CTMC) to represent and analyze software system with software aging (see Figure 4.5). The system first starts in a highly robust state S_0 and then goes into a failure probable state S_P due to software aging. Subsequently, the system goes into failure state S_F and it goes back to the state S_0 after recovery. These events are assumed to be exponentially distributed, hence such behavior may be represented by a CTMC and the events are depicted by the respective event rates. The state transition rates leaving the states S_P, S_F and S_0 are r_2, λ and r_1, respectively. The solution of the CTMC yields

the steady-state probability of S_F, which represents the steady-state unavailability of the system.

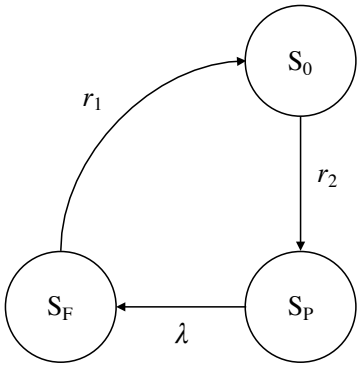


Figure 4.5 CTMC representing a system with software aging [44]

Huang et al. introduce an additional state SR for software rejuvenation as shown in Figure 4.6. The system goes into S_R from S_P when software rejuvenation is applied, and then returns back to S_0 after its completion. The rate to S_R and S_0 are specified by r_4 and r_3, respectively.

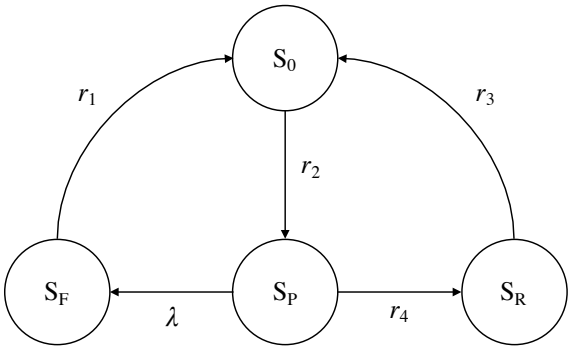


Figure 4.6 CTMC representing a system with software aging and rejuvenation [44]

The steady-state unavailability of the system can be computed by the sum of the steady-state probabilities for S_F and S_R, that is

$$U = \frac{\frac{\lambda}{r_1} + \frac{r_4}{r_3}}{1 + \frac{\lambda}{r_1} + \frac{\lambda}{r_2} + \frac{r_4}{r_2} + \frac{r_4}{r_3}}.$$

To examine the unavailability when r_4 changes, we may take the derivative of the above expression

$$\frac{dU}{dr_4} = \frac{\lambda}{r_1 r_2 r_3} \times \frac{1}{\left(1 + \frac{\lambda}{r_1} + \frac{\lambda}{r_2} + \frac{r_4}{r_2} + \frac{r_4}{r_3}\right)^2} \times \left[r_1\left(1 + \frac{r_2}{\lambda}\right) - r_3\right].$$

Since the denominator in the derivative is always positive, the sign of the derivative is determined by the sign of the term $[r_1\,(1 + r_2/\lambda) - r_3\,]$. When $r_3 > r_1\,(1 + r_2/\lambda)$, the derivative is negative implying that the unavailability always decreases as r_4 increases. Thus, for minimizing the unavailability, the rejuvenation should be performed as soon as the system enters into S_P, i.e. $r_4 = \infty$. On the other hand, when $r_3 \leq r_1\,(1 + r_2/\lambda)$, the derivative is positive or equal to 0 implying that the unavailability always increases or do not change as r_4 increases. In this case, the unavailability is minimized at $r_4 = 0$ implying that rejuvenation is not necessary. As a result, the effectiveness of rejuvenation in terms of system unavailability does not depend on the rejuvenation rate r_4, but it is determined by the threshold condition $r_3 > r_1\,(1 + r_2/\lambda)$. This is the first analytical results obtained through Markov model for software rejuvenation.

The previous CTMC model was generalized to a continuous-time semi-Markov process (SMP) by Dohi, Popstojanova, and Trivedi [53][54]. As already mentioned, in CTMC, all the transition time distributions are exponential. SMP allows us to assign general distribution to state transitions and thus extending the model capability. For example, when we consider time-based software rejuvenation, the time to trigger rejuvenation should be deterministic rather than exponentially distributed. Figure 4.7 shows the SMP with four states for representing the dynamics of software aging and rejuvenation. Although the state definitions are inherited from the original CTMC model, the state transition times are assumed to be generally distributed and all the states are considered as regeneration points. Let $F_f(t)$ and $F_r(t)$ be the distribution functions for failure time and the time to trigger software rejuvenation from failure probable state.

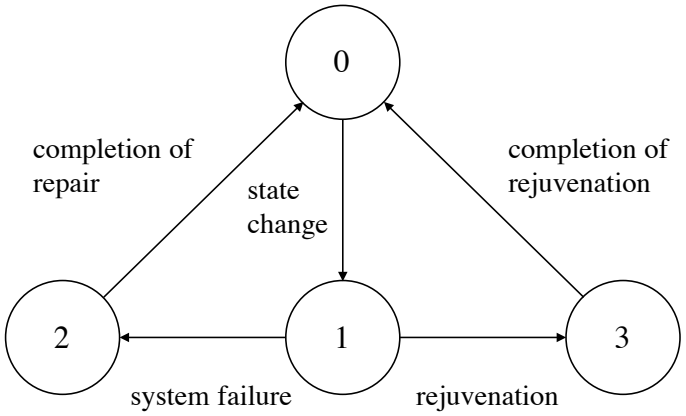


Figure 4.7 SMP with four states for representing software aging and rejuvenation [54]

In time-based software rejuvenation, $F_r(t)$ can be defined as a unit step function as bellow

$$F_r(t) = \mathrm{u}(t - t_0) = \begin{cases} 1, & \text{if } t \geq t_0 \\ 0, & \text{otherwise,} \end{cases}$$

where t_0 represents the fixed time to trigger software rejuvenation. Assume that μ_0, μ_a and μ_c represent the finite mean time to reach the probable state from the robust state, the mean time to repair, and the mean time to complete rejuvenation, respectively. The steady-state availability of the system can be given by the sum of the steady-state probabilities of the robust state and the failure-probable state and it can be derived by the standard solution technique for SMP.

$$A(t_0) = \frac{\mu_0 + \int_0^{t_0} \left(1 - F_f(t)\right) dt}{\mu_0 + \mu_a F_f(t_0) + \mu_c \left(1 - F_f(t_0)\right) + \int_0^{t_0} \left(1 - F_f(t)\right) dt} = \frac{T(t_0)}{S(t_0)}.$$

When we consider the steady-state availability of the system is the function of t_0, we can analyze the optimal rejuvenation schedule t_0^* that maximizes the system availability. Dohi, Popstojanova, and Trivedi [54] show the following theorem for optimal rejuvenation schedule.

Theorem: Suppose that the failure time distribution is strictly IFR (increasing failure rate) and $\mu_a > \mu_c$ holds. Define the following nonlinear function:

$$q_A(t_0) = T(t_0) - \{(\mu_a - \mu_c)r_f(t_0) + 1\}S(t_0),$$

where

$$r_f(t_0) = \frac{1}{1 - F_f(t_0)} \frac{dF_f(t)}{dt}.$$

(i) If $q_A(t_0) > 0$ and $q_A(\infty) < 0$, then there exists a finite and unique optimal software rejuvenation schedule t_0^* $(0 < t_0^* < \infty)$ satisfying $q_A(t_0^*) = 0$, and the maximum steady-state availability is

$$A(t_0^*) = \frac{1}{(\mu_a - \mu_c)r_f(t_0^*) + 1}.$$

(ii) If $q_A(t_0) \leq 0$, then the optimal software rejuvenation schedule is $t_0^* = 0$, and the maximum steady-state availability is $A(0) = \mu_0(\mu_0 + \mu_c)$.

(iii) If $q_A(\infty) \geq 0$, then the optimal rejuvenation schedule is $t_0^* \to 0$, and the maximum steady-state availability is $A(\infty) = (\mu_0 + \lambda_f)/(\mu_0 + \mu_a + \lambda_f)$.

Since it is quite natural to assume IFR for software aging phenomena, most probably there exists an optimal software rejuvenation schedule in practice. This result opens up further research motivations to investigate the optimal software rejuvenation in different context and various types of systems.

Xie, Hong and Trivedi [55] exploit SMP to analyze a two-level software rejuvenation policy. The SMP shown in Figure 4.7 only considers the single-level rejuvenation in which all the applications running on the system become unavailable by system level rejuvenation. However, it is not always necessary to restart the entire system for recovering erroneous application. Two-level rejuvenation approach distinguishes service-level partial rejuvenation from the system-level full rejuvenation. In the service-level partial rejuvenation, service will save any necessary data before the rejuvenation and the operation is resumed in a more efficient state after the rejuvenation. The behavior of the system is modeled by SMP as depicted in Figure 4.8.

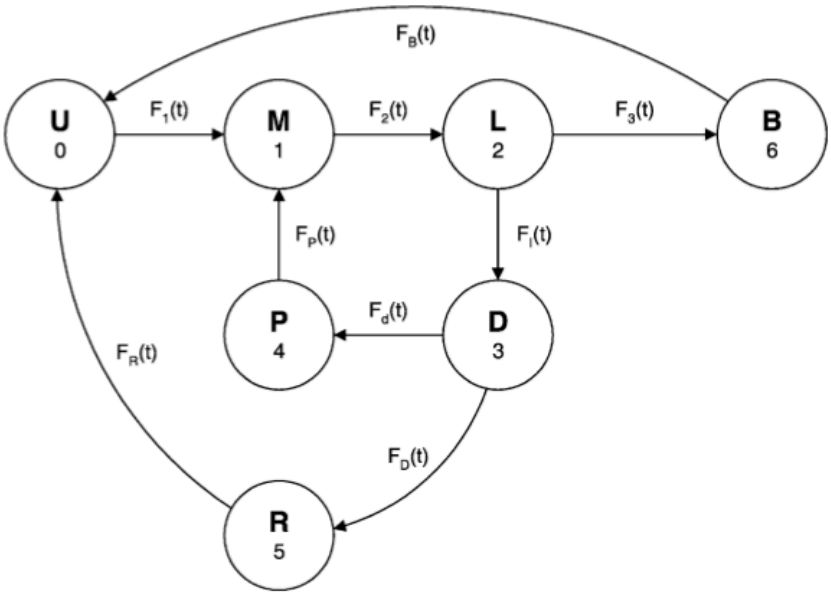


Figure 4.8 Semi-Markov process model for two-level software rejuvenation [55]

Starting from the robust state U, the system enters in the medium-efficient state M, and subsequently goes into the low-efficient state L according to the progress of software aging. From state L, the system may fail which corresponds to the transition to the reboot state B or perform rejuvenation at the decision state D. It is assumed that the transition time from state L to state B is distributed by $F_3(t)$. In case partial rejuvenation is chosen, the system enters in the partial rejuvenation state P and returns to the state M. Otherwise, the system goes full rejuvenation state R and returns back to the state U. Although the SMP model is further complex than the model in Figure 4.7, the standard two-stage solution method is still viable and the steady-state availability of the system is obtained by the sum of steady-state probabilities of state U, M and L which are denoted as π_0, π_1 and π_2, respectively. Therefore, the steady-state availability is a function of the time to trigger software rejuvenation (r), and the probability to choose partial rejuvenation (p). The steady-state availability is given by

$$A(r,p) = \pi_0 + \pi_1 + \pi_2 = \frac{S(r,p)}{S(r,p) + V(r,p)},$$

where

$$S(r,p) = \big(1 - p + pF_3(r)\big)t_0 + t_1 + \int_0^r \big(1 - F_3(t)\big)\,dt,$$

$$V(r,p) = \big(p - pF_3(r)\big)t_4 + \big(1 - p - F_3(r) + pF_3(r)\big)t_5 + F_3(r)t_6\,,$$

and t_i $(i = 0, 1, 4, 5, 6)$ is the mean sojourn times for state *i*. The optimal values of *r* and *p*, which maximize the steady-state availability, can be analyzed by taking the first-order partial derivative of $A(r,p)$ with respect to *r* and *p*. By enumerating all (*r, p*) pairs that are at boundaries and/or the first order partial derivatives are zero, it is shown that the maximum availability is achieved at either *p*=0 or 1, considering the failure rate is either an IFR, a DFR or a CFR, and $t_6 > t_5 > t_4$ holds. Note that the model is reduced into the SMP model depicted in Figure 4.8, when *p*=0, i.e., there is no partial rejuvenation.

SMP model was also adopted for analyzing optimal preventive maintenance by Chen and Trivedi [56]. The model is not limited to software aging and rejuvenation, but it is generally applicable for any preventive maintenance technique. Figure 4.9 shows the three state SMP model representing a preventive maintenance system.

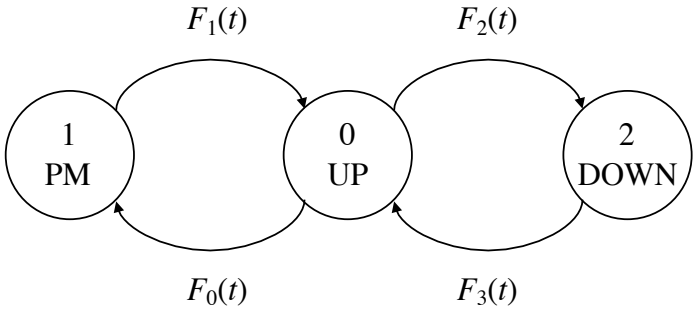


Figure 4.9 SMP model representing a preventive maintenance system [56]

State 0 represents a robust state in which the system is up. State 1 is the state where the system is under preventive maintenance and is not available. State 2 is down state and is under repair process. Let $F_0(t)$, $F_1(t)$, $F_2(t)$ and $F_3(t)$ be the distribution functions for the times to preventive maintenance, to complete the preventive maintenance, to failure, and to recover from failure, respectively. The expected sojourn times h_i in state *i* are given by

$$h_0 = \int_0^\infty \big(1 - F_0(t)\big)\big(1 - F_2(t)\big)dt,$$

$$h_1 = \int_0^{\infty} \big(1 - F_1(t)\big)dt,$$

$$h_2 = \int_0^{\infty} \big(1 - F_3(t)\big)dt.$$

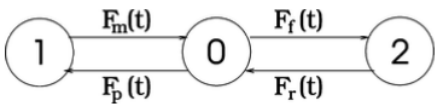
The steady-state availability of the system as a function of the time to maintenance t_0 is given by

$$A(t_0) = \frac{h_0}{h_0 + \big(1 - F_2(t_0)\big)h_1 + F_2(t_0)h_2}.$$

Similar to the theorem presented before, there exists a unique optimal time to trigger preventive maintenance when $h_1 < h_2$ and $F_2(t)$ is IFR. The three state SMP model is considered as a reduced and the simplest version of time-based software rejuvenation and hence it is used as a reference building block of further comprehensive models [57][59][60][61].

Bao, Sun and Trivedi [57] utilize the SMP model to build a hierarchical model for analyzing software aging and rejuvenation in software systems suffering from performance degradation caused by memory-leak. The hierarchical model consists of a non-homogeneous Markov chain, which is used to represent the performance degradation process and an SMP for representing software rejuvenation. The three-state SMP is used as a top-level of the hierarchical model and its failure rate is computed in a lower-level non-homogeneous Markov chain. Figure 4.10 shows the components of the hierarchical model.

Higher-level SMP



Lower-level performance degradation model

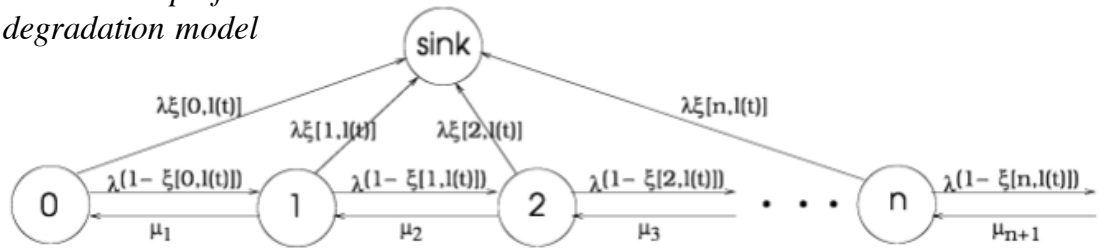


Figure 4.10 Hierarchical model for software aging and rejuvenation [57]

In the lower-level performance degradation model, the state labeled k $(k \geq 0)$ represents the system state, where k independent application processes hold a portion of memory resource. The resource requests arrive from a Poisson source with rate λ. A request is granted when sufficient memory is available, otherwise the system fails, which is represented by a transition to the sink state. The coefficient $\xi[k, l(t)]$ is the conditional probability that the system fails in state k upon the arrival of a new request while the amount of leaked resource at t in the system is $l(t)$. The holding memory at state k is released at the rate μ_k. Let $\pi(t) = [\pi_0(t), \pi_1(t), \dots, \pi_n(t), \pi_{\text{sink}}(t)]$ be the transient state probability vector, the system failure rate can be expressed as

$$h(t) = \frac{\frac{d\pi_{\text{sink}}\big(t, l(t)\big)}{dt}}{1 - \pi_{\text{sink}}\big(t, l(t)\big)} = \frac{\lambda \sum_k \xi[k, l(t)]\pi_k\big(t, l(t)\big)}{\sum_k \pi_k\big(t, l(t)\big)}.$$

This failure rate is applied for the higher-level SMP to characterize the distribution function for system failure $F_f(t)$. In the SMP, software rejuvenation is performed at the fixed time interval τ, and hence the distribution function for the time to trigger software rejuvenation is given by $F_p(t) = \mathrm{u}(t - \tau)$, which is the unit step function at τ. By the analytical result in [56], the steady-state system availability is given by

$$A(\tau) = \frac{h_0}{h_0 + \Big(1 - F_f(\tau)\Big) h_1 + F_f(\tau) h_2},$$

where $F_f(\tau) = \pi_{\text{sink}}(\tau)$ and

$$h_0 = \tau\big(1 - \pi_{\text{sink}}(\tau)\big) + \lambda \int_0^\tau t \sum\nolimits_k \xi[k, l(t)]\pi_k(t)\, dt,$$

$$h_1 = \int_0^\infty \big(1 - F_m(t)\big) dt,$$

$$h_2 = \int_0^\infty \big(1 - F_r(t)\big) dt.$$

When resource leak exists, the failure rate increases monotonically [57], and hence the analysis of steady-state system availability as the function of τ will yield the optimum software rejuvenation schedule.

While non-homogeneous Markov chain is used to represent memory leaking in the above case, Garg, Puliafito, Telek and Trivedi [58] use non-homogeneous CTMC to represent the state of transaction process in a transaction based software systems. Depending on the severity of software aging, software failure rate and service time performance are affected. Let $\mu(\cdot)$ and $\rho(\cdot)$ be the general functions for transaction service rate and system failure rate, respectively. These functions could simply depend on the time (i.e., $\mu(t), \rho(t)$), or can be the functions of instantaneous load on the system, or mean accumulated work done by the software system in given time interval, or their combination. In any case, the transaction processing state can be captured by a non-homogeneous CTMC as shown in Figure 4.11. A transaction arrives from a Poisson source with rate λ and it is processed at service rate $\mu(\cdot)$, which depends on aging state. The process is terminated when system encounters a failure or rejuvenation is triggered in accordance with a specific policy. By tracking the number of transactions in the system, the impact of rejuvenation on the probability of the transaction losses and the expected response time can be computed.

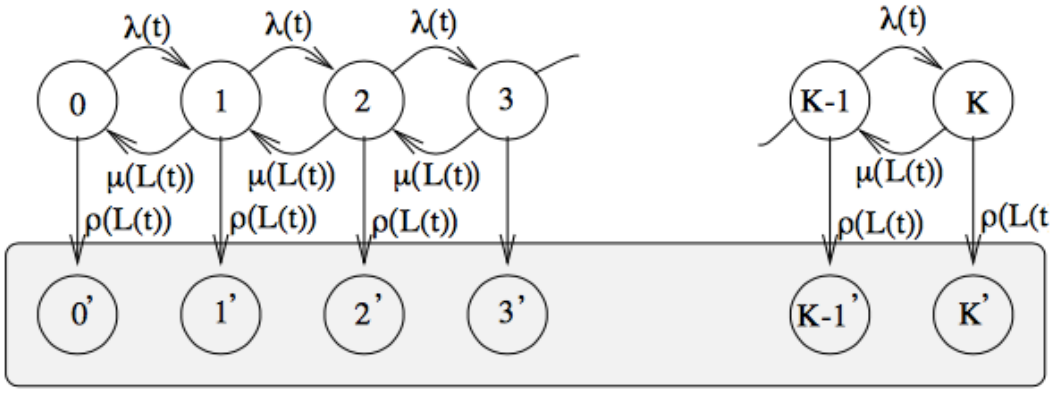


Figure 4.11 Non-homogeneous CMTC for transaction processing system with aging [58]

Hierarchical model can be used to combine analytical model with statistical models constructed from real observations of system workloads. Vaidyanathan and Trivedi [59] presented a comprehensive model for software rejuvenation in which a lower-level semi-

Markov reward model is constructed from workload and resource usage data collected from UNIX operating system, and a higher-level SMP is used to analyze the availability of the system. The statistical model captures the dynamics of resource usage and the time to resource exhaustion is estimated by the model. The estimated time to resource exhaustion is then applied to the SMP model for availability analysis. In order to build a semi-Markov reward model from real monitoring data, Vaidyanathan and Trivedi [59] use a clustering method and obtain 8 different clusters of data points. Assuming that each cluster represents a workload state of the system, the transition probability from state *i* to state *j* can be estimated from the sample data by

$$p_{ij} = \frac{observed\ no.of\ transitions\ from\ state\ i\ to\ state\ j}{total\ observed\ no.of\ transitions\ from\ state\ i}.$$

On the other hand, the sojourn time distribution for each state is fitted either 2-stage hyper-exponential or 2-stage hypo-exponential distributions. While the transition probability matrix $P = (p_{ij})$ and the sojourn time distributions can characterize the SMP for workload state transition, the resource usage in each state is incorporated by assigning reward to the state. The estimated slopes for *usedSwapSpace* and *realMemoryFree* are assigned as reward rates to the states and they are used to compute the estimated time to resource exhaustion. In the higher-level SMP, the distribution function of time to failure is approximated by 2-stage Erlang distribution, whose mean value is given by the estimated time to resource exhaustion. Since Erlang distribution is an IFR distribution, the optimal software rejuvenation schedule can be obtained by the analysis of the SMP.

Although the analysis of optimal software rejuvenation schedule relies on the distribution function of the time to system failure, in practice such a distribution is not easy to acquire due to the lack of failure data. Accelerated life test (ALT) for software aging [38] provides a powerful solution to this issue by estimating the lifetime distribution from the lifetime data obtained under accelerated stresses. Zhao, Jin, Trivedi and Matias Jr. [60] first applies ALT to web application system where memory leak fault is injected and the failure time distribution is fitted to the Inverse Power Law (IPL) Weibull distribution and IPL lognormal distribution. The IPL Weibull distribution is given by

$$f(t,s) = \beta k s^w (k s^w t)^{\beta-1} e^{-(k s^w t)^\beta},$$

where β is the shape parameter of Weibull distribution, k and w are IPL model parameters, and s is the stress level of ALT. The values of β, k and w can be estimated by experiments [61]. The estimated IPL Weibull distribution is then used as the failure time distribution in the three state SMP model (Figure 4.9), and the optimal software rejuvenation schedule can be obtained through the analysis of the SMP.

Based on the SMP model presented in [53], Okamura and Dohi [46] extend the model to consider the opportunity time-triggered rejuvenation policy, where software rejuvenation performs at the first opportunity after the scheduled trigger time of rejuvenation. The policy is proposed to address the problem of time-based software rejuvenation in which software rejuvenation is performed without considering operational condition. In practice, the application may not be stopped depending on the operational state. The chance to perform rejuvenation after the scheduled rejuvenation time is called opportunity. In order to incorporate such an opportunity waiting condition, the SMP model is extended as shown in Figure 4.12.

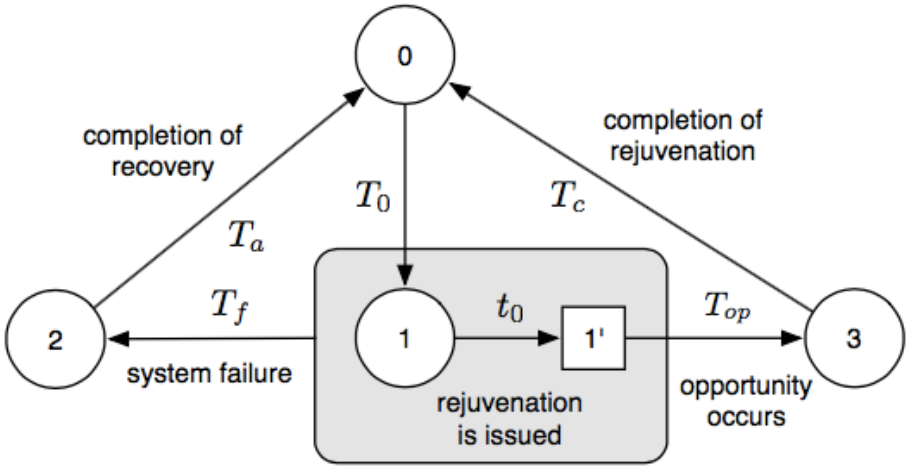


Figure 4.12 State transition diagram for opportunity-based rejuvenation model [47]

In addition to four states in the original model, the state of failure probable and waiting for opportunity is inserted as state 1'. The system can fail from either state 1 or state 1', but the rejuvenation will be performed only after visiting state 1'. T_{op} represents the duration for

waiting the rejuvenation opportunity and the distribution function for T_{op} has dependency on the time to trigger rejuvenation t_0 as can be expressed as $F_{op}(s; t_0)$. When we assume the opportunity will occur by following a renewal process, which is independent of the system behavior, $F_{op}(s; t_0)$ is given by

$$F_{op}(s; t_0) = \frac{1}{E[Y]} \int_0^s \big(1 - G(t)\big) dt,$$

where Y is the random variable representing the time interval between consecutive opportunities with the distribution function $G(t)$. $F_{op}(s; t_0)$ is essentially the equilibrium distribution in the renewal process theory [62]. In this case, it is proved that there exists the optimal trigger time of rejuvenation if failure time distribution is IFR [47]. Similarly, Okamura and Dohi [47] show that the sufficient condition for the existence of the optimal trigger time under the opportunity time-triggered rejuvenation policy, where the opportunity process follows an independent Markovian Arrival Process.

Machida, Nicola and Trivedi [63] introduced SMP to model software aging and rejuvenation in a server virtualized system and analyze the completion time of the job running on the system. Job completion time is another important performance measure of software system, while many of previous studies focus on steady-state system availability or downtime costs. To compute the job completion time, the progress of job execution on the system needs to be tracked along with the state transitions of the system. In [63], the system state transition is modeled by an SMP and the distribution of job completion time on the system is computed through the framework presented by Kulkarni, Nicola and Trivedi [64]. Figure 4.13 shows the SMP model for a virtualized server with Cold-VM rejuvenation in which the execution of the VM is stopped and the running job needs to restart from its beginning.

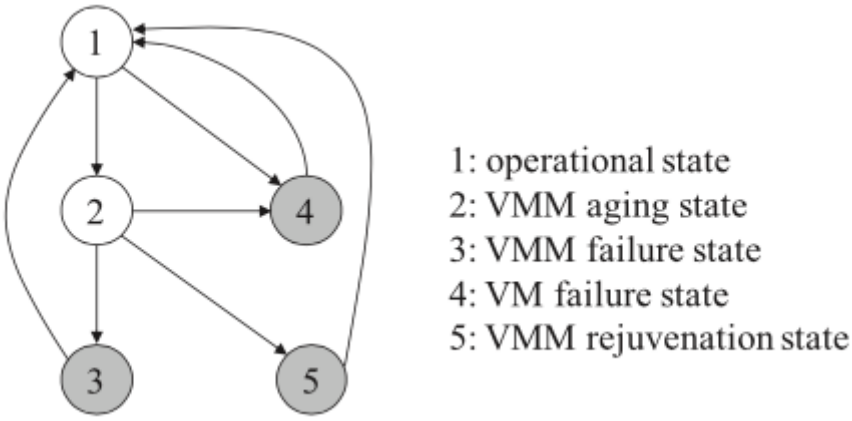


Figure 4.13 SMP model for a virtualized server with Cold-VM rejuvenation [63]

The VMM failure in State 3, VM failure in State 4, and VMM rejuvenation in State 5 are categorized as preemptive-repeat (*prt*) states, which are represented as shaded circles. When the system enters in a *prt* state, all the intermediate results of job execution is lost and the job needs to be restarted from the beginning. As a result of the analysis using the framework, the job completion time distribution is derived in the form of Laplace-Stieltjes transform (LST).

$$F^{\sim}(s) = \int_0^{\infty} e^{-st} dF(t),$$

where $F(t), t > 0$ is the original distribution function. For Cold-VM rejuvenation case, the LST of the job completion time is given by

$$F_{cold}^{\sim}(s) = \sum_{k=1}^{5} L_k^{\sim}(s,x) \cdot \pi_k,$$

$$L_i^{\sim}(s,x) = \begin{cases} M_i^{\sim}(s,x), & i = 1,2 \\ Y_i^{\sim}(s) M_1^{\sim}(s,x), & i = 3,4,5, \end{cases}$$

where π_k is the steady-state probability that the system is in state k, $M_i^{\sim}(s,x)$ is the LST of the job completion time conditioned by starting upon entry to state i, and $Y_i^{\sim}(s)$ is the LST of the residual time in state i. The approximate original distribution can be obtained by numerical inversion technique.

4.2.1.3. Stochastic Petri nets

Markov models allow us to capture the fundamental behavior of software aging and rejuvenation as presented in the literature. Analytic solution of such models gives solid theoretical results such as the condition where a finite and unique optimal rejuvenation schedule exists. However, as the scale of software systems grow, the needed size of state spaces expands exponentially. The scalability issue of Markov models arises. Complex large-scale systems are not easily modeled by Markov chains and the solution to large-scale Markov chains often require formidable efforts. A promising approach to deal with the scalability issue is to employ a higher-level representation of stochastic processes for system state transitions. Stochastic Petri nets (SPN) is a variant of Pet nets and is often used as a higher-level representation of stochastic processes including Markov models. The definition of SPN is given by the five-tuple $SPN = (P, T, A, \mu_0, \Lambda)$,

- $P = \{p_1, p_2, \cdots, p_m\}$ is a set of places (drawn as circles).
- $T = \{t_1, t_2, \cdots, t_n\}$ is a set of transitions (drawn as bars).
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs connecting *P* and *T*.
- $\mu: P \rightarrow \{0,1,2, \cdots\}$ is the marking that denotes the number of tokens for each place in *P*. The initial marking is denoted as μ_0.
- $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ is an array of firing rates associated with transitions.

The firing time of transition t_i is assumed to be exponentially distributed with rate λ_i. SPN model can be transformed into an equivalent CTMC by enumerating possible markings and transitions among them. In general, SPN gives more compact representation of complex and large-scale state-space models. In the following, we will review the applications of SPN to model software aging and rejuvenation for complex systems.

The initial attempt to apply SPN to model software rejuvenation system was carried out by Garg, Puliafito, Telek, and Trivedi [52]. In order to deal with the deterministic transition for rejuvenation trigger, Markov regenerative Stochastic Petri Net (MRSPN), which allows immediate and generally-distributed firing time, was used. Although underlying stochastic

process is not a CTMC, it falls into a class of Markov regenerative process (MRGP) that can be analyzed by Markov renewal theory. Figure 4.14 shows the MRSPN for time-based software rejuvenation system.

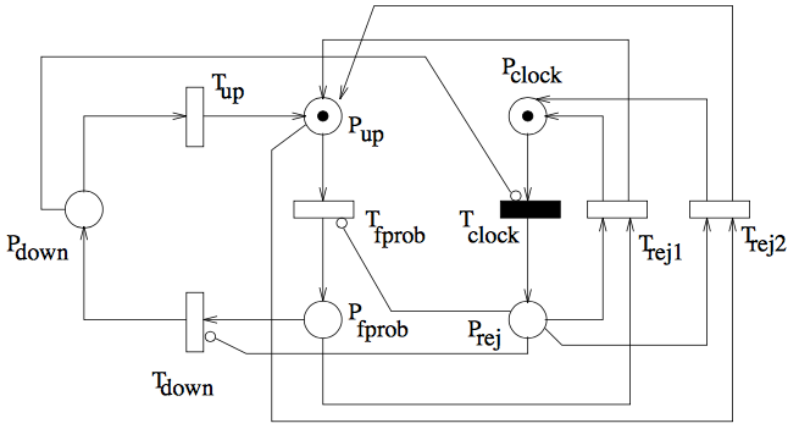


Figure 4.14 MRSPN model for software rejuvenation [52]

The places P_{up}, P_{fprob}, and P_{down} represent the robust, failure-probable and down local state of the system, respectively. Meanwhile, the transitions T_{fprob}, T_{down} and T_{up} represent the state transitions to P_{fprob}, P_{down}, and P_{up}, whose transition rates are exponentially distributed. Time-based software rejuvenation is represented by the deterministic transition T_{clock} depicted as filled rectangular. A token in P_{clock} is removed when the fixed time interval passes and subsequently a token is deposited in the place P_{rej}, representing the rejuvenation state. From MRSPN in Figure 4.14, the reachability graph can be obtained as shown in Figure 4.15, where the label of each state indicates the net markings in (P_{up}, P_{fprob}, P_{down}, P_{clock}, P_{rej}). For example, (10010) means one token is stored in places P_{up} and P_{clock}, which represents the up state. Since the reachability graph has a non-regeneration marking that is labeled (01010), the graph is essentially an MRGP. By solving the MRGP, we can obtain the steady-state probability of each marking. The steady-state availability of the system can be computed by the sum of steady-state probabilities for the markings (10010) and (01010).

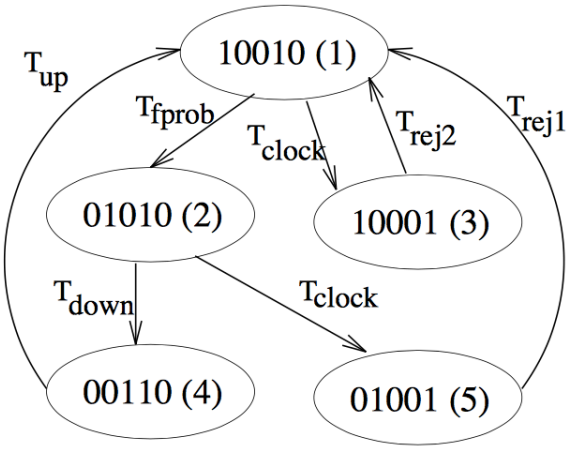


Figure 4.15 Reachability graph for the MRSPN model [52]

It is worth noting that this time-based software rejuvenation model can be reduced into three state SMP model, but the better expressiveness originated from SPN opens up further exploration of comprehensive modeling for complex and large-scale systems.

Bobbio et al. [65] introduced a Fluid Stochastic Petri Nets (FSPN) to model a system using checkpointing, software rejuvenation and self-restoration.

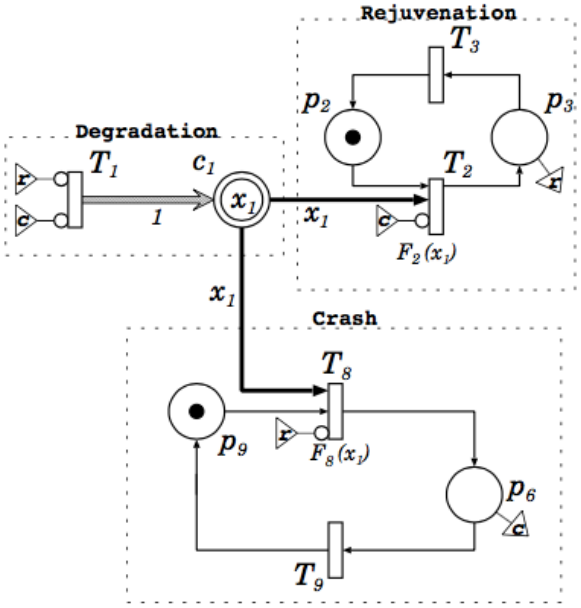


Figure 4.16 FSPN model for software rejuvenation system [65]

FSPN extends SPN such that the model can capture the continuous flow of transitions. Since software aging can be regarded as error accumulation process, it ca be modeled by a continuous flow to the continuous state. By FSPN, the behavior of software rejuvenation can be represented as Figure 4.16, which could be a component of the whole system. The subnet labeled degradation consists of transition T_1 and continuous place c_1, whose marking x_1 represents the degradation level. Transition T_1 pumps fluid in place x_1 and represents the increasing of the system degradation level unless there is token in p_3 or p_6. In the subnet labeled rejuvenation, the transitions T_2 and T_3 represent the trigger and completion of rejuvenation, respectively. The firing rate $F_2(x_1)$ depends on the fluid level in c_1. Alternatively, the subnet labeled crash represents the system failure and recovery process. Transition T_8 and T_9 correspond to the system failure and recovery transitions, respectively. The firing rate $F_8(x_1)$ depends on the fluid level in c_1. Those subnets are further integrated with the subnets of workload, checkpointing and self-restoration [65].

So far, rejuvenation for a single instance of software process has been considered. When multiple instances of software executions are independently subject to aging and forced to perform rejuvenation, the state-space becomes huge. Vaidynanathan, Harper, Hunter and Trivedi [66] model and analyze software rejuvenation in cluster systems by using Stochastic Reward Nets (SRNs), which introduce reward assignment to SPN. In SRN, every tangible marking can be associated with a reward rate. It can be shown that an SRN is mapped into a Markov Reward model and thus a variety of reward-based measures can be specified and calculated by SRN. The software rejuvenation model for the cluster systems consisting of n nodes are presented in Figure 4.17. At the beginning, all the n nodes are in the robust state, which is represented by tokens in place P_{up}. As time goes by, each node eventually transits to a failure-probable state (place P_{fprob}) represented the transition T_{fprob} firing and subsequently goes fail (place $P_{nodeFail}$). A node can be successfully repaired with probability c or can fail with a probability (1-c), leading to a system failure (all n nodes are down). Time-based rejuvenation is represented by the deterministic transition $T_{rejinterval}$, which fires every d time units. Its firing stores a token in place $P_{startrejuv}$. If there are tokens in places P_{up} or P_{fprob}, immediate transitions T_{immd8} and T_{immd9} are enabled respectively. All the clustered nodes are

rejuvenated in a sequence. After a node has been rejuvenated, it goes back to the robust state, represented by tokens in place $P_{rejuved}$.

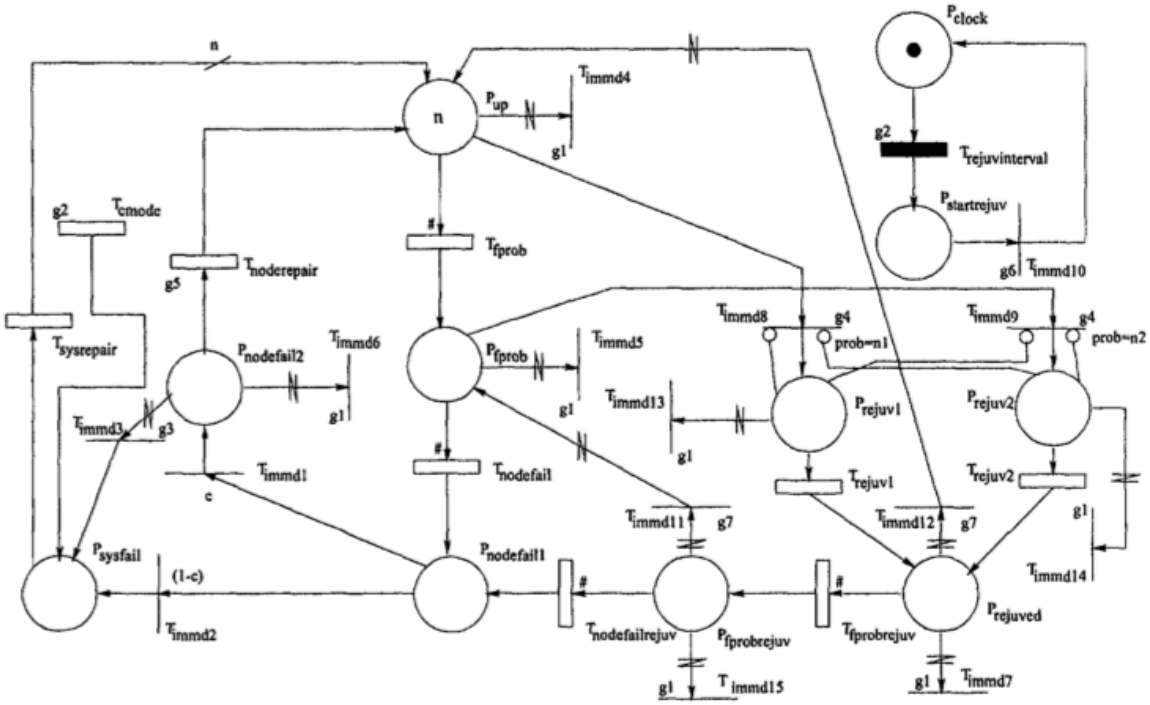
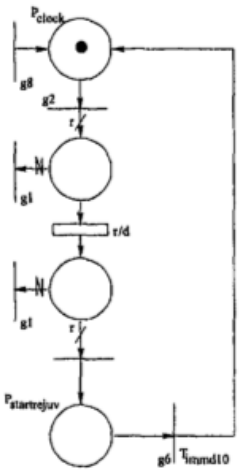


Figure 4.17 SRN for cluster system employing time-based rejuvenation [66][67]

Rejuvenation of the cluster completes when all *n* nodes are rejuvenated, when the immediate transition T_{immd10} becomes enabled. The deterministic transition in the clock subnet can be approximated by an r-stage Erlang distribution. Each stage of the Erlang distribution is exponentially distributed with mean *d*/*r*. The approximated SRN with the associated guard functions are shown in Figure 4.18. The reward functions to evaluate the effects of cluster rejuvenation are given in Table 4.1.

Table 4.1 Reward functions [66][67]

Unavailability	$(\#P_{sysfail}==1)\ ?\ 1 : 0$
Cost	$\#P_{rejuv} * cost_{rejuv} + \#P_{nodefail2} * cost_{nodefail} + \#P_{sysfail} + cost_{sysfail}$



Guard	Function
g_1	$(\#P_{sysfail} == 1)\ ?\ 1 : 0$
g_2	$(\#P_{sysfail} != 1)\ ?\ 1 : 0$
g_3	$(\#P_{nodefail2} + \#P_{rejuv1} + \#P_{rejuv2} == a)\ ?\ 1 : 0$
g_4	$(\#P_{startrejuv} == 1)\ ?\ 1 : 0$
g_5	$(\#P_{startrejuv} != 1)\ ?\ 1 : 0$
g_6	$(\#P_{nodefail2}+\#P_{rejuved}+\#P_{fprobrejuv} == n \ \|\| \ \#P_{sysfail} == 1 \ \|\| \ \#P_{nodefail2} == a\text{-}1)\ ?\ 1 : 0$
g_7	$(\#P_{clock} == 1)\ ?\ 1 : 0$
g_8	$(\#P_{sysfail} == 1\ \&\&\ (\#P_{startrejuv} + \#P_{clock} == 0))\ ?\ 1 : 0$

Figure 4.18 R-Stage Erlang approximation of deterministic transition and guard functions [66][67]

The expected unavailability is computed as the expected number of tokens in place $P_{sysfail}$, while the expected cost can be computed as the sum of the costs due to rejuvenation, node failures, and system failures. With these measures, the optimal rejuvenation interval can be obtained by using solution tool Stochastic Petri Net Package (SPNP) [68].

For cluster systems offering web application service, the performance of the service is important as well as service availability. Performability measures can be useful for quantification of such services. The rejuvenation impact on the performability of a cluster system was analyzed by Wang, Xie and Trivedi [69] by taking into account workload intensity and fluctuation. Assuming that the workload intensity changes among peak periods and off-peak periods in deterministic time interval, Deterministic and Stochastic Petri Net (DSPN) is employed. DSPN is a class of MRSPN such that each transition in the model is either exponentially distributed or deterministic, besides adopting immediate transitions. While the cluster system is modeled by SRN similar to the one in [66], the DSPN model introduces an additional subnet for workload status as shown in Figure 4.19. A token is deposited in either P_{peak} or $P_{offpeak}$, which represents the peak period or off-peak period, respectively. Transitions T_{peak} and $T_{offpeak}$ fire in deterministic time interval, and thus the entire model becomes a DSPN.

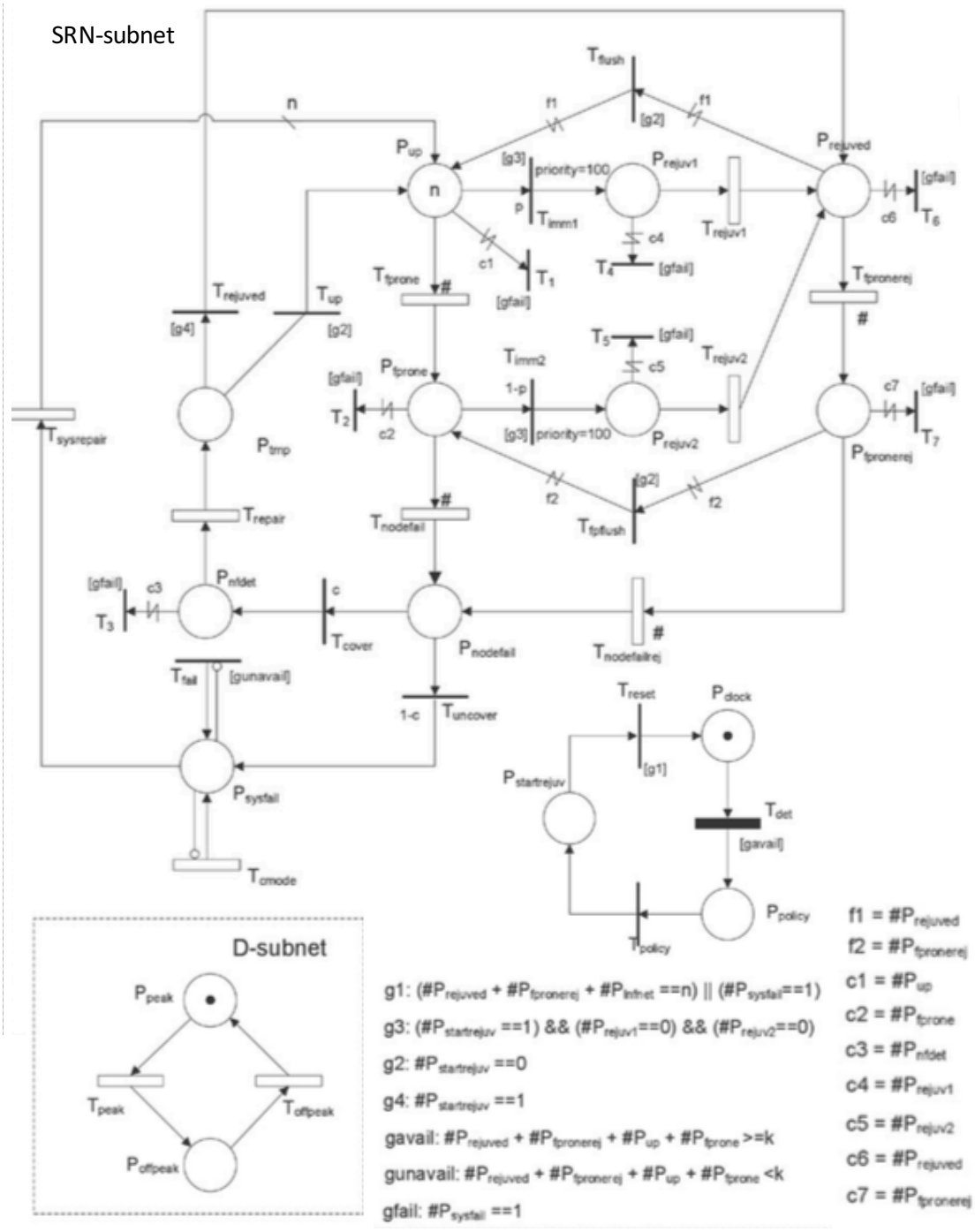


Figure 4.19 DSPN for cluster rejuvenation system with periodic workload changes [69]

Depending on the workload status, request arrival rates are different, and also impacts the failure rate. To analyze the performability, the performance of clustered system is modeled

as a *M*/*M*/*i*/*m*+*i* queue (see Figure 4.20), where *i* is the number of servers in the system, and *m* is the maximum queue length.

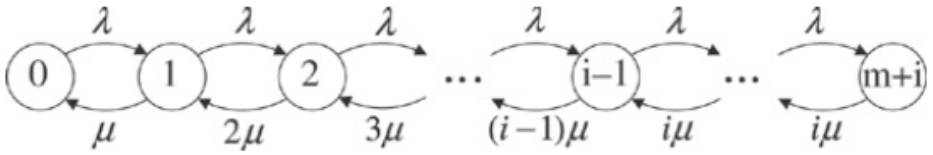


Figure 4.20 A *M*/*M*/*i*/*m*+*i* queue [69]

The parameter values are supplied by the corresponding marking of the DSPN model. For marking *c* of the DSPN model, *i* is computed as the sum of the number of tokens in P_{up} or $P_{rejuved}$, P_{fprone}, and $P_{fpronerej}$.

$$i(c) = \#\mathrm{P}_{\mathrm{up}}(c) + \#\mathrm{P}_{\mathrm{rejuved}}(c) + \#\mathrm{P}_{\mathrm{fprone}}(c) + \#\mathrm{P}_{\mathrm{fpronerej}}(c).$$

Request arrival rate (λ) changes depending on the workload state as

$$\lambda(c) = \begin{cases} \lambda_{\mathrm{peak}}, & \text{if } \#P_{\mathrm{peak}}(c) = 1 \\ \lambda_{\mathrm{offpeak}}, & \text{if } \#P_{\mathrm{pffpeak}}(c) = 1. \end{cases}$$

μ_{up} and μ_{fp} denote the service rates for up servers and failure-prone servers, respectively. The equivalent service rate μ for the marking *c* of the DSPN is computed as

$$\mu(c) = \frac{\mu_{\mathrm{up}}\left(\#P_{\mathrm{up}}(c) + \#P_{\mathrm{rejuved}}(c)\right) + \mu_{\mathrm{fp}}\left(\#P_{\mathrm{fprone}}(c) + \#P_{\mathrm{fpronerej}}(c)\right)}{i}.$$

The blocking probability and system throughput can be computed from the *M*/*M*/*i*/*m*+*i* model [70]. The blocking probability is

$$P_b(c) = \frac{\lambda(c)^{m+i}}{i^m \cdot i! \cdot \mu(c)^{m+i}} \cdot \frac{1}{\sum_{k=0}^{i} \frac{\lambda(c)^k}{k! \cdot \mu(c)^k} + \sum_{k=i+1}^{i+m} \frac{\lambda(c)^k}{i^{k-i} \cdot i! \cdot \mu(c)^k}},$$

and the throughput is given by $T_{\mathrm{put}}(c) = \lambda(c) \cdot \left(1 - P_b(c)\right)$. The results of numerical experiments imply the following general policy for rejuvenation scheduling. If the rejuvenation is scheduled early in peak period, the best rejuvenation policy is to perform rejuvenation immediately. Such a strategy can reduce the job blocking probability, and

increase the system throughput. Otherwise, it should be delayed until the next off-peak period begins [69].

Nowadays, server virtualization becomes popular to build server systems by virtual machines (VMs). Since a VM is essentially instantiated as a software process on top of server visualization software, both of VM and virtualization software can entail risk of software aging. Machida, Kim and Trivedi [12][71] investigate the software aging and rejuvenation techniques for server virtualized systems and model the system behavior using SRN. By leveraging the modularity of SRN formalism, three different rejuvenation methodologies are modeled simultaneously; namely Cold-VM rejuvenation, Warm-VM rejuvenation and Migrate-VM rejuvenation. Figure 4.21 shows SRN model for a server virtualized system with Cold-VM rejuvenation.

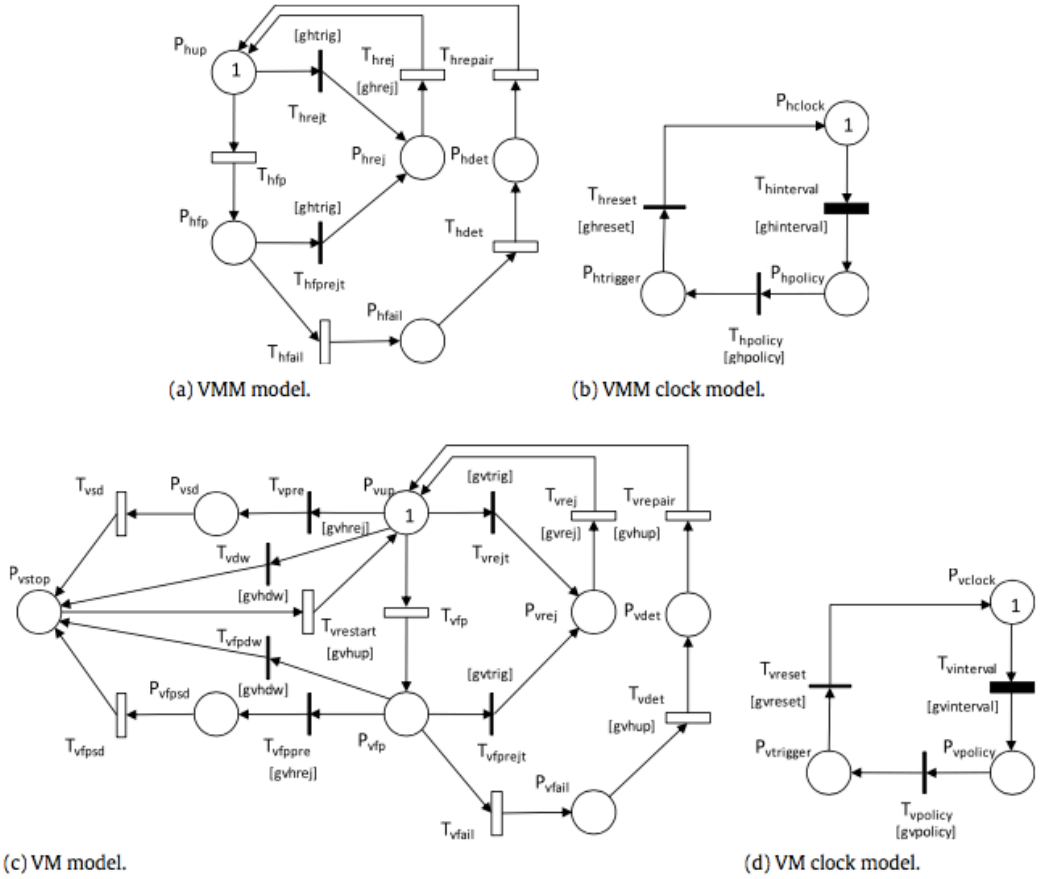


Figure 4.21 SRN model for a server virtualized system with Cold-VM rejuvenation [12]

The model consists of four subnets. Virtual machine monitor (VMM) model represents the software aging and rejuvenation process of a host server with a VMM. The VMM software rejuvenation for is triggered by the timer represented by the clock model. VM model represents the state transition of a VM running on the host. The execution of VM will stop when the underlying VMM goes rejuvenation or fail. Meanwhile, the VM also has risk of software aging and software rejuvenation is applied by following to the clock represented by VM clock model. Thus, system availability is affected by the two rejuvenation parameters; VM rejuvenation trigger interval and VMM rejuvenation trigger interval. As numerical example in Figure 4.22 shows, there exists the optimal combination of those parameters, which maximizes the steady-state availability.

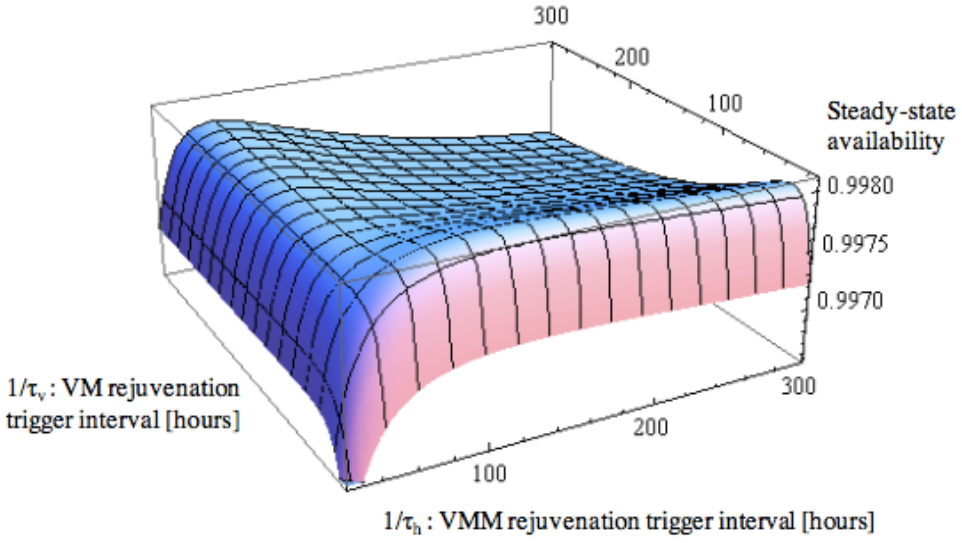


Figure 4.22 Steady-state availability of the system with Cold-VM rejuvenation [12]

In Cold-VM rejuvenation, the VM running on the host is also stopped at VMM rejuvenation, and it can restart after the rejuvenation process. On the other hand, in Warm-VM rejuvenation, the VM is suspended before VMM rejuvenation, and its execution is resumed after rejuvenation conclusion. It is intuitive that Warm-VM rejuvenation is better for high-availability of VM. However, comparative numerical experiments with sensitivity analysis on rejuvenation trigger interval of VM reveals that Cold-VM rejuvenation technique achieves

higher VM availability when the interval is long. The effectiveness of VMM rejuvenation with live VM migration is also extensively studied in Machida, Kim and Trivedi [12].

Melo, Pualo, Araujo, Matos and Araujo [72] also consider the combination of live VM migration and VMM rejuvenation in cloud computing environment. In order to consider the availability of management server which controls VM migration, Reliability Block Diagram (RBD) is integrated hierarchically with SPN subnets as shown in Figure 4.23. The model consists of higher-level SPN and lower-level RBD. Two RBDs are used to compute the failure and repair times of the management server and node, which are input to the parameter values of SPN. The first subnet of SPN represents the failure and recovery process of the management server, meanwhile the second subnet is the clock model for triggering VMM rejuvenation. The third subnet represents the state transitions of the VM running on either main node or standby node. Those models are used to analyze the steady-state availability of the system, where SHARPE [73] and TimeNET [74] are used to evaluate the RBD and SPN, respectively.

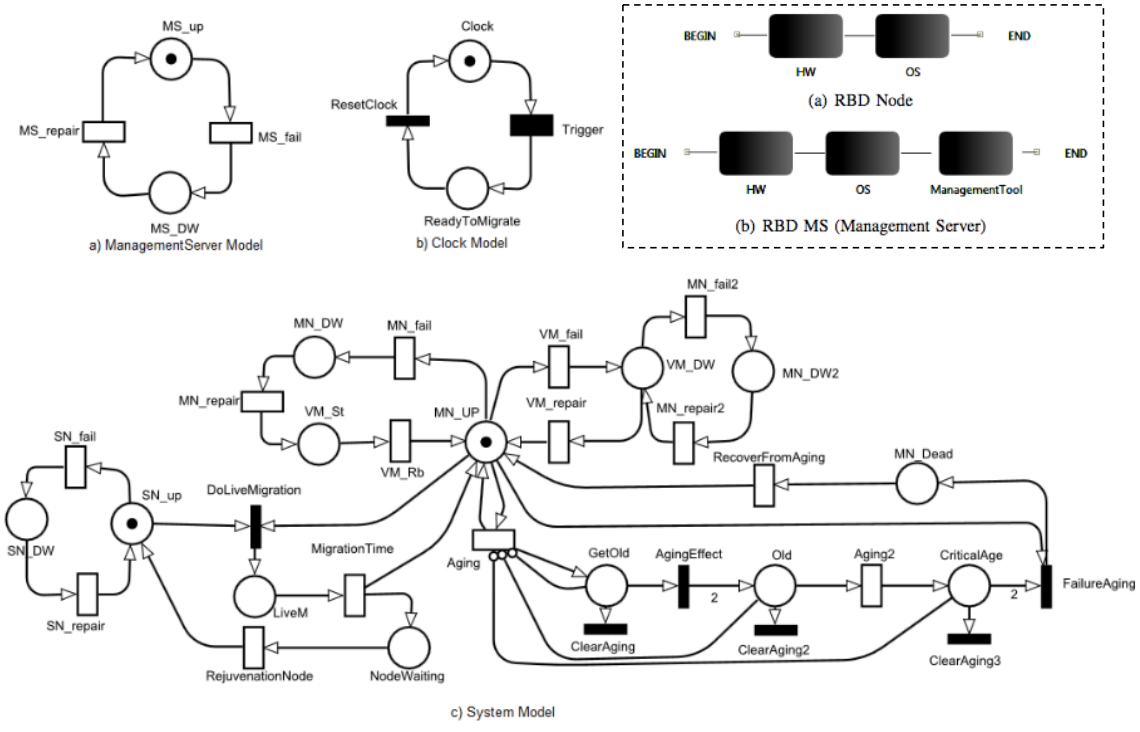
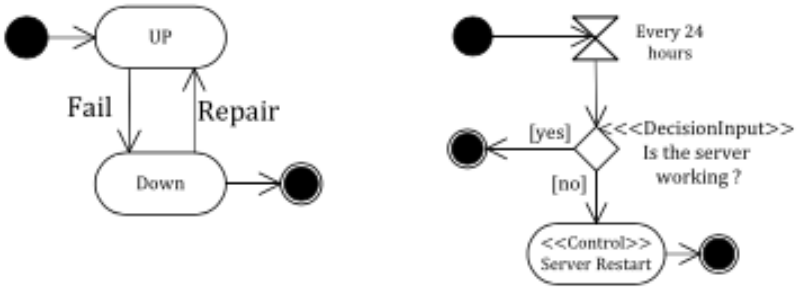


Figure 4.23 Hierarchical model for virtualized system with live VM migration [72]

Okamura, Guan, Duo and Dohi [10] analyze the resiliency of a virtualized system with Cold-VM rejuvenation and Warm-VM rejuvenation. Resiliency is defined as the persistence of service delivery that can justifiably be trusted, when facing changes [8] and a resiliency quantification technique is presented by [9]. In order to quantitatively evaluate the resiliency of a virtualized system, the SRN models in [12] are transformed into an approximated CTMC by PH-expansion. As a result of numerical study, it is shown that the time duration of change is insensitive to the system availability and the system with Cold-VM rejuvenation is more robust for the change [10].

Another high-level representation of a system is given by semi-formal specification languages, such as Unified Modeling Language (UML) [75], and Systems Modeling Language (SysML) [75]. Although SRN helps the specification process when compared with CTMC, there are still huge gap between the capabilities of system engineers and the required skill to correctly specify SRNs. Andrade, Machida, Kim and Trivedi [78] propose an automated transformation method from SysML model to SRN, and apply the method to analyze a system with software rejuvenation. In SysML, state transition diagram can be used to represent state transitions of a system component (such as software aging, failure and recovery). On the other hand, system operations such as monitoring, failure detection and rejuvenation are described by activity diagram. Figure 4.24 shows an example of state machine diagram for a server process and an activity diagram representing server maintenance operation.



State machine diagram for a server process

Activity diagram for a server maintenance operation

Figure 4.24 State machine diagram and activity diagram for software maintenance [78]

According to the defined translation rules, those diagrams are transformed into the corresponding SRNs as shown in Figure 4.25.

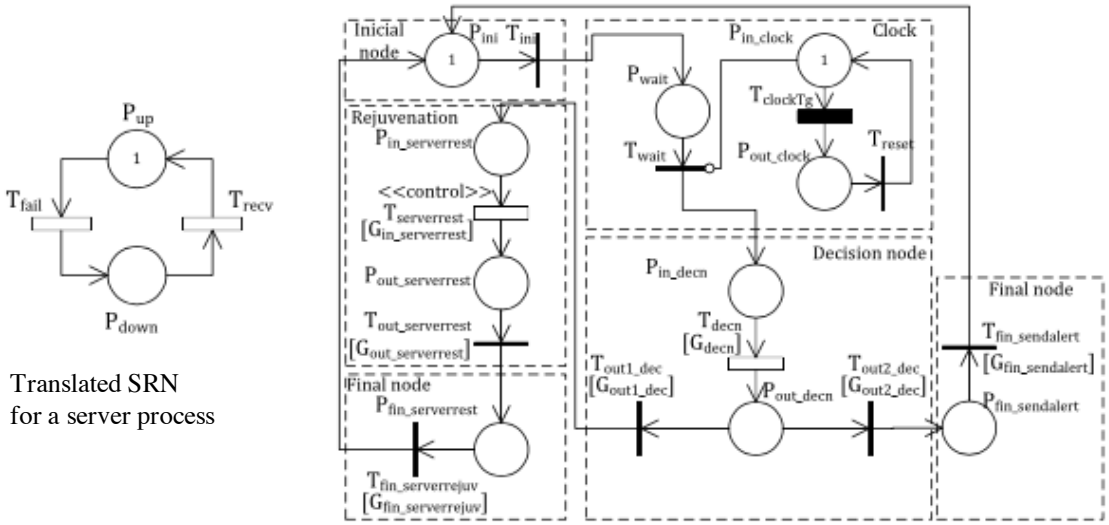


Figure 4.25 Translated SRNs for server process and server maintenance operation [78]

The SRN subnets transformed by SysML diagrams are integrated adopting the technique presented in [77]. The generated SRN models for condition-based rejuvenation and time-based rejuvenation were validated with the corresponding conventional models studied in [44] and [52], respectively.

4.2.1.4. Summary

This section reviewed the evolution of state-space models for software rejuvenation originated from the CTMC presented by Huang et al [44]. Two main streams, which continue from the original to the recent studies on software rejuvenation are outlined: 1) the extension of Markov models and 2) the utilization of higher-level representation. The former thread includes the approach with SMP, which can relax the assumption of exponential distribution in Markov models and allows the analysis of the optimal rejuvenation schedule in time-based software rejuvenation scheme. Hierarchical models give powerful solution to integrate the rejuvenation model with a comprehensive model for software aging process. On the other

hand, the latter thread often resorts to the expressive power of SPN that can be transformed into equivalent Markov chains in order to capture more complex and larger systems. SPN-based models are applied to software rejuvenation in fault-tolerant system, cluster systems, and server virtualized systems. System availability and other performance measures can be computed by reward assignments to SPN.

4.2.2. Optimal stopping without job arrival

This section describes the research contribution that formulated software rejuvenation decision problem in a job processing system as an optimal stopping problem and derived the analytical solution to the problem to minimize the performance cost [49]. The work has been done in collaboration with Prof. Naoto Miyoshi in Tokyo Institute of Technology.

In this study, we derived the optimal policy for deciding software rejuvenation trigger in a job processing system when observing a performance degradation of the system. At the detection of service degradation, the system is assumed to enter a digesting phase where newly arrival jobs are rejected and queued jobs are processed with decreased service rate. The service can be modeled as a pure death process, and both of the delayed jobs due to decreased service rate, the dropped jobs caused by rejuvenation, and the rejected arrival jobs are counted as the cost components. We formulated the decision problem of software rejuvenation, which trade-offs the costs associated with the decisions, as an optimal stopping problem. The analysis on the optimality equation clarifies the conditions to switch the optimal policy for triggering software rejuvenation to minimize the expected cost.

4.2.2.1. Introduction

Deciding the optimal trigger of software rejuvenation is an important issue for offering better system performance and availability. A number of research works have been presented to model the behavior of software system and analyze the impact of software rejuvenation. Most of conventional studies on software rejuvenation model leverage state-space models including Continuous-time Markov Chain [44], Markov Regenerative Stochastic Petri net [58] and semi-Markov process [54] etc. that capture the state transitions of target system and

find the optimal interval to trigger rejuvenation in terms of system availability or performance. These state-space models are useful for analyzing the system in its design phase. However, software rejuvenation in the operational phase could be treated as a dynamic decision problem that decides when to trigger software rejuvenation depending on the observed system state. In this case, the strategy of software rejuvenation is called condition-based instead of time-based. The optimal decision of condition-based software rejuvenation in terms of steady-state system availability has been studied in the literature [79][80]. In this section, we do not consider system failures caused by aging that impacts on system availability. Instead we focus on the performance trade-offs affected by condition-based software rejuvenation.

In this section, we address a dynamic decision problem of software rejuvenation and formulate the problem as an optimal stopping problem. The optimization goal is set to minimize the performance loss of a job processing system whose service is modeled as an M/M/1 queue. We assume a non-fatal error causes degradation of service performance that results in increased number of delayed jobs. The number of delayed jobs could incur the cost. Software rejuvenation is applied to clear the error condition and restore the desirable performance, while system down caused by software rejuvenation drops the jobs processing in the system. Therefore, there is a trade-off between the number of job affected by delayed service and the number of dropped jobs. Since the trade-off is determined by the trigger of rejuvenation, deciding the rejuvenation trigger to minimize the cost is the issue to be concerned.

The rest part of the section is organized as follows. Section 4.2.2.2 describes the target system to be analyzed and provides the problem statement. Section 4.2.2.3 formulates the decision problem of software rejuvenation as an optimal stopping problem. Section 4.2.2.4 shows a set of propositions and corollaries that gives the optimal policy for applying software rejuvenation under specific conditions. Section 4.2.2.5 shows a numerical example and Section 4.2.2.6 gives a summary of this section.

4.2.2.2. Problem statement

Consider a job processing system that provides a service to process jobs requested from service users. Job requests arrive continuously, and the service executes the requested jobs in an FCFS manner as long as the system is available. From the users' perspective, the job execution performance and the service availability are important criteria to evaluate the system. When the system falls into an error condition that degrades the service performance, the completion of the requested job might be prolonged. If the service degradation is too serious to continue the operation, the system operator might decide to kill the processing jobs and recover the system with the original performance by rejuvenation. On the other hand, if the performance degradation is not so serious such that the remaining job can finish the execution in a reasonable delay, the trigger of rejuvenation might be postponed. Therefore, the decision of software rejuvenation trigger impacts on job performance and needs to be determined carefully in consideration with the number of remaining jobs and degraded performance. In contrast to the related work like [81], our study does not assume the degradation level goes worse after the detection.

4.2.2.3. Problem formulation

For the system in a normal condition, we assume the job requests arrive by Poisson process with rate λ and service time is exponentially distributed with rate μ. Thus, the system in normal state is modeled by M/M/1 queue. The traffic intensity $\rho = \lambda/\mu$ is assumed to be less than 1. After sufficiently long period of time, the performance is degraded due to a system error. We assume the service performance is degraded to $r\mu$ $(0 < r < 1)$. When the system acknowledges the error condition, it starts rejecting the newly arrival jobs to prepare for rejuvenation. The remaining jobs in the system keep being processed with the degraded service rate until system goes rejuvenation. Given an error condition, the system operator or controller has two options; i) perform rejuvenation immediately to restore the original performance and ii) wait the completions of remaining jobs and postpone the decision of rejuvenation trigger by τ. We denote these actions a_r and a_w, respectively. The choice of decision is made repeatedly until rejuvenation is performed eventually. When the number of jobs in the system becomes 0, the operator performs rejuvenation in the next decision point.

The above system state transitions can be captured by a state model with the set of state S that includes the initial state N_0 and normal states with *i* processing jobs N_i, and error state with i processing jobs E_i, and a rejuvenation state R. In this paper, we focus on the rejuvenation transitions that correspond to the state transitions from state E_i to R. Figure 4.26 shows the state transition diagram where the labels on the arcs represent the state transition rates of exponential distributions.

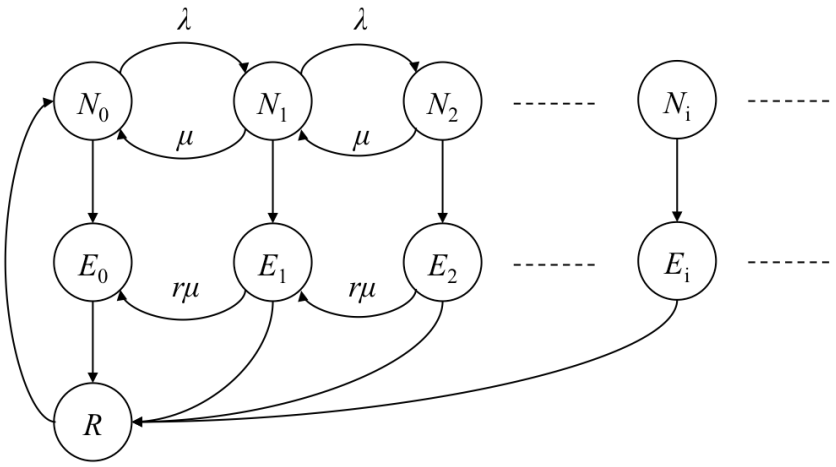


Figure 4.26 State transition diagram for the job processing system [49]

When the system enters in an error state and rejuvenation is not performed immediately, the state transition follows a pure death process with service rate $r\mu$, since the new requests are blocked. The transition probability from state *i* to state *j* ($\leq i$) by the action a_w is given by

$$P_{i,j}(a_w) = \begin{cases} e^{-r\mu\tau}\dfrac{(r\mu\tau)^{i-j}}{(i-j)!} & (1 \leq j \leq i) \\ 1 - \displaystyle\sum_{k=0}^{i-1} e^{-r\mu\tau}\dfrac{(r\mu\tau)^{k}}{k!} & (j = 0). \end{cases} \quad (1)$$

When software rejuvenation trigger is decided in an error state, the transition probability is given by $P_{i,R}(a_r) = 1$. After the rejuvenation, the system returns back to the initial state and hence the transition probability is $P_{R,N_0} = 1$.

As mentioned previously, the trigger of software rejuvenation is determined in consideration with the cost. We formulate the cost function associated with the defined state transitions. There are three types of costs to be considered. The first cost takes into account

the number of jobs rejected due to service unavailability. In error states and rejuvenation state, the newly arriving jobs are rejected. The associated cost is proportional to the length of service unavailable period. The second cost addresses the number of dropped jobs due to rejuvenation. All the remaining jobs in the system are dropped when software rejuvenation is performed. The third cost takes into account the number of delayed jobs that complete the execution after service degradation. Although all these costs are associated with the number of affected jobs, the impact assignment might be different by users. Hence, we introduce coefficients to individual cost categories, c_{reject}, c_{drop} and c_{delay}, respectively. We assume the cost of job drop is more expensive than the cost of job delay; $c_{\text{drop}} > c_{\text{delay}}$. The definitions of cost coefficients are summarized in Table 4.2.

Table 4.2 Definitions of cost coefficients [49]

Coefficient	***Definition***
c_{reject}	Cost of rejected jobs which arrive in error or rejuvenation states
c_{drop}	Cost of dropped jobs which are in the system when rejuvenation is triggered
c_{delay}	Cost of delayed jobs due to decreased service rate in error states

Using the cost coefficients, the cost functions for possible actions in error states is defined as follows:

$$\begin{aligned} C_{i,R}(a_r) &= c_{\text{drop}} \cdot i \\ C_{i,j}(a_w) &= c_{\text{delay}} \cdot (i-j) + c_{\text{reject}} \cdot \lambda\tau \quad (j < i). \end{aligned} \tag{2}$$

When the operator decides the rejuvenation in state *i*, *i* jobs are sacrificed for rejuvenation. When the decision of rejuvenation trigger is postponed by τ and the remaining job is changed from *i* to *j*, *i-j* jobs are counted as the delayed jobs.

The problem to determine when to start software rejuvenation given an error state can be formulated as an optimal stopping problem. After the detection of an error state E_i, the number of jobs in the system gradually decreases by the pure death process. Either a_r or a_w is chosen at every decision points. The decision process is stopped when a_r is chosen,

while the process continues by taking action a_w. The optimality equation of this problem is given by

$$v(i) = \begin{cases} \min\left[C_{i,R}(a_r), \sum_{j=0}^{i} P_{i,j}(a_w)\{C_{i,j}(a_w) + v(j)\}\right], & i > 0 \\ 0, & i = 0 \end{cases} \quad (3)$$

Whenever the cost incurred by rejuvenation, $C_{i,R}(a_r)$ becomes less than the expected cost resulted from the action a_w, the action a_r is chosen and the decision process stops.

4.2.2.4. Analysis of optimal policy

This section provides the analysis of the optimal stopping problem for software rejuvenation in a job processing system and derives the optimal policy that maps state E_i to the optimal action in the state. In particular, we show that under the specific condition the action a_r is always chosen after the number of jobs becomes less than or equal to the value i^*, otherwise the action a_w is chosen.

Let $\varphi(i)$ be the expected cost of the action a_w in E_i.

$$\begin{aligned} \varphi(i) &= \sum_{j=0}^{i} P_{i,j}(a_w)\{C_{i,j}(a_w) + v(j)\} \\ &= \sum_{j=0}^{i} P_{i,j}(a_w)\{c_{\text{delay}} \cdot (i-j) + c_{\text{reject}} \cdot \lambda\tau \ + v(j)\} \\ &= \sum_{j=0}^{i} P_{i,j}(a_w) \cdot c_{\text{delay}} \cdot (i-j) + P_{i,i}(a_w)v(i) + \sum_{j=0}^{i-1} P_{i,j}(a_w)v(j) + c_{\text{reject}} \cdot \lambda\tau. \end{aligned}$$

By definition, a_w is chosen whenever the state is E_i. Thus, replacing $v(i)$ by $\varphi(i)$ and define $\psi_i \coloneqq \left[1 - P_{i,i}(a_w)\right]^{-1}$, we have

$$\varphi(i) = \psi_i \left\{ \sum_{j=0}^{i} P_{i,j}(a_w) \cdot c_{\text{delay}} \cdot (i-j) + \sum_{j=0}^{i-1} P_{i,j}(a_w)v(j) + c_{\text{reject}} \cdot \lambda\tau \right\}.$$

Substituting variable j to $k = i - j$,

$$\varphi(i) = \psi_i \left\{ \sum_{k=0}^{i} P_{i,i-k}(a_w) \cdot c_{\text{delay}} \cdot k + \sum_{k=1}^{i} P_{i,i-k}(a_w) v(i-k) + c_{\text{reject}} \cdot \lambda\tau \right\}$$

$$= \psi_i \left\{ \sum_{k=1}^{i} P_{i,i-k}(a_w) \left\{ c_{\text{delay}} k + v(i-k) \right\} + c_{\text{reject}} \lambda\tau \right\}.$$

Therefore, we redefine the optimality equation for $i > 0$ as follows:

$$v(i) = \min\left[c_{\text{drop}} \cdot i, \varphi(i) \right]$$

$$\varphi(i) = \psi_i \left\{ \sum_{k=1}^{i} P_{i,i-k}(a_w) \left\{ c_{\text{delay}} k + v(i-k) \right\} + c_{\text{reject}} \lambda\tau \right\} \quad (4)$$

In order to compare the expected costs of the two decision options, we start from showing the following lemma.

Lemma 4.2.2.1. Define $\chi_i = \sum_{k=1}^{i} P_{i,i-k}(a_w) \cdot k$ for $i > 0$.

a) χ_i is monotonically increasing in *i*, and

b) $\lim_{i \to \infty} \chi_i = r\mu\tau$.

Proof. By the definition of the transitions probabilities of the pure death process (1), $P_{i,0}(a_w)$ satisfies

$$P_{i,0}(a_w) = 1 - \sum\nolimits_{k=0}^{i-1} e^{-r\mu\tau} \frac{(r\mu\tau)^k}{k!}$$

$$= 1 - \sum\nolimits_{k=0}^{i} e^{-r\mu\tau} \frac{(r\mu\tau)^k}{k!} + e^{-r\mu\tau} \frac{(r\mu\tau)^i}{i!}$$

$$= P_{i+1,0}(a_w) + P_{i+1,1}(a_w)$$

and $P_{i,i-k}(a_w) = P_{i+1,i+1-k}(a_w)$ for $\forall k, 1 \le k \le i-1$. The difference of χ_i is then computed by

$$\chi_{i+1} - \chi_i = P_{i+1,0}(a_w) \cdot (i+1) + \sum\nolimits_{k=1}^{i} P_{i+1,i+1-k}(a_w) \cdot k$$

$$- \left\{ P_{i,0}(a_w) \cdot i + \sum\nolimits_{k=1}^{i-1} P_{i,i-k}(a_w) \cdot k \right\}$$

$$= P_{i+1,0}(a_w) \cdot (i+1) - P_{i,0}(a_w) \cdot i + P_{i+1,1}(a_w) \cdot i$$
$$= P_{i+1,0}(a_w) \cdot (i+1) - \{P_{i+1,0}(a_w) + P_{i+1,1}(a_w)\} \cdot i + P_{i+1,1}(a_w) \cdot i$$
$$= P_{i+1,0}(a_w) > 0.$$

Therefore χ_i is monotonically increasing in *i*.

For the limit of χ_i,

$$\chi_i = P_{i,0}(a_w) \cdot i + \sum_{k=1}^{i-1} P_{i,i-k}(a_w) \cdot k$$
$$= \left\{1 - \sum_{k=0}^{i-1} e^{-r\mu\tau} \frac{(r\mu\tau)^k}{k!}\right\} \cdot i + \sum_{k=1}^{i-1} e^{-r\mu\tau} \frac{(r\mu\tau)^k}{k!} \cdot k$$
$$= \left\{1 - \sum_{k=0}^{i-1} e^{-r\mu\tau} \frac{(r\mu\tau)^k}{k!}\right\} \cdot i + r\mu\tau \sum_{k=1}^{i-1} e^{-r\mu\tau} \frac{(r\mu\tau)^{k-1}}{(k-1)!}$$

Since

$$\sum_{k=0}^{\infty} e^{-r\mu\tau} \frac{(r\mu\tau)^k}{k!} = 1,$$

we obtain $\lim_{i\to\infty} \chi_i = r\mu\tau$. ■

The next proposition gives the condition where the action a_r is chosen at state E_i.

Proposition 4.2.2.1. If χ_i satisfies the condition $\chi_i \leq \frac{c_{\text{reject}} \cdot \lambda\tau}{c_{\text{drop}} - c_{\text{delay}}}$, the action a_r is always chosen in any error states $E_l, l \in [1, i]$.

Proof. From the lemma 1.a), $\chi_l \leq \frac{c_{\text{reject}} \cdot \lambda\tau}{c_{\text{drop}} - c_{\text{delay}}}$ for any $l \in [1, i]$. Starting from the state E_1, The expected cost for the action a_w is

$$\varphi(1) = \frac{1}{1 - P_{1,1}(a_w)} \{P_{1,0}(a_w)\{c_{\text{delay}} + v(0)\} + c_{\text{reject}} \cdot \lambda\tau\}$$
$$= c_{\text{delay}} + \frac{1}{1 - P_{1,1}(a_w)} c_{\text{reject}} \cdot \lambda\tau$$

$$= c_{\text{delay}} + \frac{1}{\chi_1} c_{\text{reject}} \cdot \lambda\tau$$

By the given condition for χ_1, $\varphi(1) \geq c_{\text{drop}}$ that yields $v(1) = c_{\text{drop}}$ by the action a_r. Next we prove that the action a_r is chosen at E_l, provided that the action a_r is always chosen for any $E_{l^-}, l^- \in [1, l-1]$. The assumption follows $v(l^-) = c_{\text{drop}} \cdot l^-$ for any l^-. Then the expected cost for the action a_w at E_l is

$$\begin{aligned}
\varphi(l) &= \psi_l \left\{ \sum_{k=1}^{l} P_{l,l-k}(a_w) \left\{ c_{\text{delay}} \cdot k + v(l-k) \right\} + c_{\text{reject}} \lambda\tau \right\} \\
&= \psi_l \left\{ \sum_{k=1}^{l} P_{l,l-k}(a_w) \left\{ c_{\text{delay}} \cdot k + c_{\text{drop}} \cdot (l-k) \right\} + c_{\text{reject}} \lambda\tau \right\} \\
&= \psi_l \left\{ c_{\text{drop}} \{1 - P_{l,l}(a_w)\} l + \sum_{k=1}^{l} P_{l,l-k}(a_w)\, k \{ c_{\text{delay}} - c_{\text{drop}} \} + c_{\text{reject}} \lambda\tau \right\} \\
&= c_{\text{drop}} \cdot l + \psi_l \{ \chi_l \cdot \{ c_{\text{delay}} - c_{\text{drop}} \} + c_{\text{reject}} \lambda\tau \}
\end{aligned}$$

By the condition for χ_l, we get $\varphi(l) \geq c_{\text{drop}} \cdot l$ that implies a_r is chosen at E_l. Since the result holds for any $l \in [1, i]$, the proposition 1 is satisfied. ■

Note that when $\chi_l = \frac{c_{\text{reject}} \cdot \lambda\tau}{c_{\text{drop}} - c_{\text{delay}}}$, $\varphi(l)$ is equal to $c_{\text{drop}} \cdot l$ that means both the actions a_r and a_w are possible at E_l.

From the Proposition 4.2.2.1 and Lemma 4.2.2.1.b, the following corollary can be obtained without proof.

Corollary 4.2.2.1. The optimal policy choses the action a_r for $E_l, \forall l \in [1, \infty)$, when

$$\frac{r}{\rho} \leq \frac{c_{\text{reject}}}{c_{\text{drop}} - c_{\text{delay}}}.$$

The proposition 4.2.2.1 shows the optimal policy for state $E_l, l \in [1, i]$. Next, we consider the policy for state $E_h, h > i$ provided that action a_w is chosen at E_{i+1}.

Proposition 4.2.2.2. If χ_i and χ_{i+1} satisfy the condition $\chi_i \leq \frac{c_{\text{reject}} \cdot \lambda\tau}{c_{\text{drop}} - c_{\text{delay}}} < \chi_{i+1}$, the action a_w is always chosen in any error states $E_h, h > i$.

Proof. By the Proposition 4.2.2.1, $v(l) = c_{\text{drop}} \cdot l$ for any $l \in [1, i]$. The expected cost for the action a_w at E_{i+1} is given by

$$\varphi(i+1) = c_{\text{drop}} \cdot (i+1) + \psi_{i+1}\{\chi_{i+1} \cdot \{c_{\text{delay}} - c_{\text{drop}}\} + c_{\text{reject}}\lambda\tau\}.$$

From the given condition for χ_{i+1}, $\varphi(i+1) < c_{\text{drop}} \cdot (i+1)$ that implies that action a_w is chosen at E_{i+1}. Suppose that $\varphi(h^-) < c_{\text{drop}} \cdot h^-$ holds for $\forall h^- \in [i+1, h-1]$. We prove that $\varphi(h) < c_{\text{drop}} \cdot h$ also holds in this case. The expected cost for the action a_w at E_h is

$$\varphi(h) = \psi_h \left\{ \sum_{k=1}^{h} P_{h,h-k}(a_w) \{c_{\text{delay}} k + v(h-k)\} + c_{\text{reject}}\lambda\tau \right\}$$

$$= \psi_h \left\{ \sum_{k=1}^{h-i-1} P_{h,h-k}(a_w) \{c_{\text{delay}} \cdot k + v(h-k)\} \right.$$

$$\left. + \sum_{k=h-i}^{h} P_{h,h-k}(a_w) \{c_{\text{delay}} \cdot k + v(h-k)\} + c_{\text{reject}}\lambda\tau \right\}$$

$$= \psi_h \left\{ \sum_{k=1}^{h-i-1} P_{h,h-k}(a_w) \{c_{\text{delay}} \cdot k + \varphi(h-k)\} \right.$$

$$\left. + \sum_{k=h-i}^{h} P_{h,h-k}(a_w) \{c_{\text{delay}} \cdot k + c_{\text{drop}} \cdot (h-k)\} + c_{\text{reject}}\lambda\tau \right\}$$

$$= \psi_h \left\{ \sum_{k=1}^{h} P_{h,h-k}(a_w)\, c_{\text{drop}} \cdot h - \sum_{k=1}^{h-i-1} P_{h,h-k}(a_w)\, c_{\text{drop}} \cdot h \right.$$

$$+ \sum_{k=1}^{h-i-1} P_{h,h-k}(a_w) \left\{ c_{\text{delay}} \cdot k + \varphi(h-k) \right\}$$

$$\left. + \sum_{k=h-i}^{h} P_{h,h-k}(a_w) \left\{ c_{\text{delay}} - c_{\text{drop}} \right\} k + c_{\text{reject}} \lambda \tau \right\}$$

$$= c_{\text{drop}} \cdot h + \psi_h \left\{ \sum_{k=1}^{h-i-1} P_{h,h-k}(a_w) \left\{ c_{\text{delay}} \cdot k - c_{\text{drop}} \cdot k - c_{\text{drop}} \cdot (h-k) + \varphi(h-k) \right\} \right.$$

$$\left. + \sum_{k=h-i}^{h} P_{h,h-k}(a_w) \left\{ c_{\text{delay}} - c_{\text{drop}} \right\} k + c_{\text{reject}} \lambda \tau \right\}$$

$$= c_{\text{drop}} \cdot h + \psi_h \left\{ \sum_{k=1}^{h-i-1} P_{h,h-k}(a_w) \left\{ -c_{\text{drop}} \cdot (h-k) + \varphi(h-k) \right\} \right.$$

$$\left. + \sum_{k=1}^{h} P_{h,h-k}(a_w) \left\{ c_{\text{delay}} - c_{\text{drop}} \right\} k + c_{\text{reject}} \lambda \tau \right\}.$$

Since $-c_{\text{drop}} \cdot (h-k) + \varphi(h-k) < 0$ for $1 \le k \le h-i-1$ and $\chi_h\{c_{\text{delay}} - c_{\text{drop}}\} < -c_{\text{reject}} \cdot \lambda\tau$ from Lemma 4.2.2.1.a, we obtain $\varphi(h) < c_{\text{drop}} \cdot h$ and thus $v(h) = \varphi(h)$. By induction, we can conclude the action a_w is always chosen in any error states $E_h, h > i$. ■

From the Proposition 4.2.2.1 and 4.2.2.2, we can derive the next corollary about the optimal policy for the rejuvenation decision.

Corollary 4.2.2.2. There is an optimal control limit i^* where the action a_r is chosen for $\forall E_l, l \le i^*$ and the action a_w is chosen for $\forall E_h, h > i^*$, when the cost coefficients satisfy the following condition

$$\frac{r}{\rho} > \frac{c_{\text{reject}}}{c_{\text{drop}} - c_{\text{delay}}}.$$

The optimal control limit i^* can be obtained by

$$i^* = \underset{l}{\text{argmin}} \left\{ \frac{c_{\text{reject}} \cdot \lambda\tau}{c_{\text{drop}} - c_{\text{delay}}} - \chi_l \geq 0 \right\}. \quad (5)$$

Corollary 4.2.2.1 and 4.2.2.2 imply that the optimal state to start rejuvenation can be determined when the degradation level *r* and the number of remained jobs in the system *i* are given.

4.2.2.5. Numerical example

Based on the analytical results obtained in the previous section, we conduct a numerical study to show the optimal policy for rejuvenation with given parameter values. In this numerical study, we use the parameter values shown in Table 4.3.

Table 4.3 Parameter values used in the numerical example [49]

parameter	***description***	***value***
c_{drop}	Cost for dropped jobs due to rejuvenation	10
c_{delay}	Cost for delayed jobs	5
λ	Job arrival rate	0.04
μ	Service rate	1
r	Service degradation level	0.5

The optimal control limit i^* is changed by varying the value of c_{reject} and decision time interval τ. The computed results are shown in Figure 2.

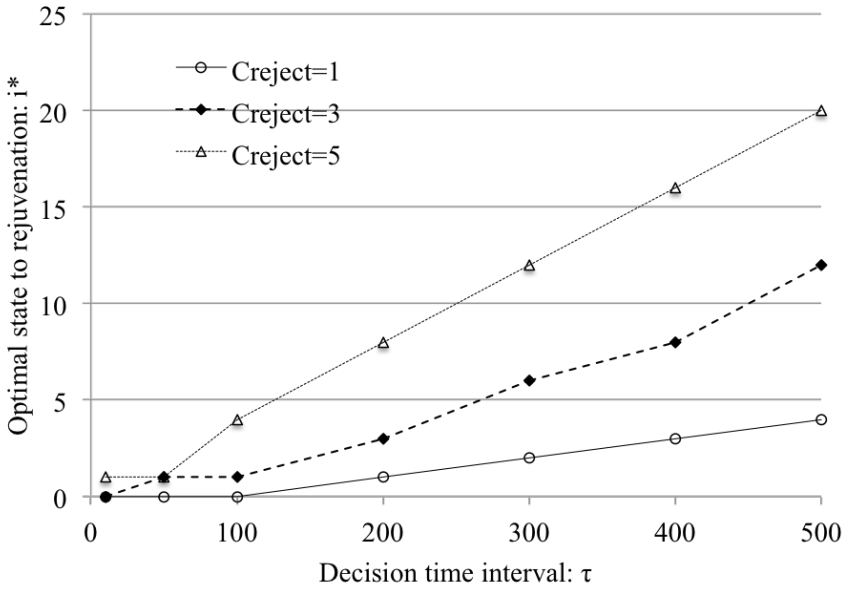


Figure 4.27 Optimal control limits by varying the decision time interval [49]

The results show that the optimal control limit i^* increases when the decision time interval becomes long. The limit also increases according to the cost for rejected jobs. The increased values of the two factors lead to the increased cost of action a_w, and hence rejuvenation decision tends to be made even when there are some waiting jobs in the queue.

4.2.2.6. Summary

In this section, we presented the optimal policy for dynamic software rejuvenation in a job processing system affected by service degradation through the analysis of optimal stopping problem. With our optimal policy, the optimal timing to trigger software rejuvenation can be determined by the observed state with the number of jobs in the system and the service degradation level. This study does not consider the risk of progressive degradation of service performance and the impacts of system failure. These aspects will be considered in the future work.

Although we modeled the service by M/M/1, the job arrival process might have seasonality in real systems. In case the arrival rate changes by seasons, the arrival rate at the entrance of error states is concerned. The expression (5) shows that the higher arrival rate λ

causes the larger value of i^*. This indicates that the higher arrival rate observed in the error states encourages earlier rejuvenation decision, which can reduce the number of rejected jobs. Further discussions about the possible impacts of different arrival processes and service time distributions on the optimal rejuvenation decision could be in the future work.

4.2.3. Optimal stopping with job arrival

This section describes the extension of the study presented in the previous section (Section 4.2.2). The job processing system considered in the previous section assumed that newly arrival jobs are rejected in the rejuvenation decision phase. The extended work relaxed this assumption so that new jobs are queued even during the rejuvenation decision process. We reformulated the problem as another optimal stopping problem and derived an analytical solution to this problem. The work has been done in collaboration with Prof. Naoto Miyoshi in Tokyo Institute of Technology and the outcome has been published in Reliability Engineering and System Safety in 2017 [50].

In this study, we theoretically derived the optimal policy that minimizes the cost of decision for software rejuvenation in a deteriorating job processing system, which is modeled as an M/M/1 queue with infinite buffer size. In our model, the number of queued jobs is used to represent the system state and the decision of rejuvenation is made upon the completion of a foreground job. We formulated the problem as an optimal stopping problem to analytically derive the optimal policy for the rejuvenation decision. The analytical results show that the optimal stopping policy is determined by the service degradation rate, the costs of dropped jobs and delayed jobs, and it does not depend on the number of queued jobs. This indicates that whether to trigger rejuvenation can be decided immediately when the system confirms the level of service degradation, regardless of the number of queued jobs at that time instant.

4.2.3.1. Introduction

Software rejuvenation in the operational phase can be dealt as a dynamic decision problem for deciding when to trigger software rejuvenation depending on the observed system states.

This strategy is called condition-based rejuvenation, which is the condition-based maintenance for software runtime. The optimal solution for condition-based software rejuvenation has been studied in the literature, where a semi-Markov decision process was used to determine the system state in which to trigger rejuvenation to maximize system availability [79][80]. These models can be used to compute the downtime cost due to software rejuvenation, while performance models may be required to take into account the impact of rejuvenation on the application performance that is the focus of the present study.

In this study, we considered condition-based rejuvenation for a deteriorating job processing system and analytically derive the optimal policy to determine the time to trigger software rejuvenation in terms of job completion time performance. Examples of job processing system modeled in this study include web application services and HPC applications. We model the system as an M/M/1 queue and formulate the decision problem for software rejuvenation as an optimal stopping problem. In the job processing system, the rejuvenation cost is proportional to the number of queued jobs. The number of queued jobs decreases as jobs are processed, while newly arriving jobs increase the queue length. Until rejuvenation, the system performance degrades which affects the completion time performance of the queued jobs. The affected jobs are considered as the cost, in particular when the service provider specifies the service level for users on the job processing performance. The rejuvenation trigger needs to be determined in consideration of this cost trade-off. When a job execution is completed, the system user is notified and has the option to trigger rejuvenation or to continue processing jobs. Once the software rejuvenation is applied, software aging is not observed again for a long time. Hence, we focus on one-time decision of rejuvenation and its impacts on the running jobs on the system. We formulate the problem as an optimal stopping problem and prove that the optimal decision for the rejuvenation time depends on the level of service degradation, and the costs associated with the rejuvenation and delayed job processes, while it is independent of the number of queued jobs in the system. Unlike conventional studies on condition-based software rejuvenation, we focus on one-time decision of rejuvenation (i.e, stopping problem) and do not rely on any

numerical methods for deriving the optimal decision policies but analytically prove the optimality of our decision policy.

The rest of the section is organized as follows. Section 4.2.3.2 describes related work. Section 4.2.3.3 explains the job processing system to be modeled and introduces the necessary assumptions. We define the optimal stopping problem for deciding the optimal trigger for rejuvenation. Section 4.2.3.4 analyzes the problem in detail and mathematically proves the optimal policy for performing rejuvenation. We also show that the derived policy does not depend on the number of queued jobs. Finally, Section 4.2.3.5 gives a summary.

4.2.3.2. Related work

As reviewed in Section 4.2.1, a number of studies conduct the analysis of optimal interval for time-based software rejuvenation to maximize the system availability or performance. The state-space modeling approaches aim to determine the optimal periodicity for software rejuvenation. Under the time-based software rejuvenation policy, the system is interrupted periodically at predetermined intervals without knowing the current system state. On the other hand, if the system states are observable precisely at any decision time, condition-based software rejuvenation can determine the trigger timing for software rejuvenation in response to the observed state.

For the condition-based software rejuvenation approach, Pfening et al. introduced a Markov decision model to determine the optimal time to rejuvenate a server in terms of rejuvenation cost [81]. Chen et al. used a semi-Markov decision process (SMDP) to deal with the joint optimization problem to derive the optimum inspection rate and rejuvenation policy [79]. The SMDP approach was also used in [80] to analyze the optimal policy for condition-based rejuvenation for multistage degradation software. These studies take into account the trade-off between rejuvenation cost and system down cost, while the costs associated with job interruption and delayed processing are not investigated.

Okamura et al. present a Markov decision process for determining software rejuvenation in a transaction-based system [82]. The transaction process is modeled as a Markovian arrival process (MAP) and power efficiency is also considered in the performance criteria to be

optimized. With the state definition including the phase of the MAP, the degradation level and the number of transactions, the optimality equation is formulated. Numerical experiments show the monotone policy with respect to the level of degradation and the number of transactions. Compared to [82], we deal with a simpler problem, to show analytically the optimal policy for rejuvenation in a deteriorating job processing system.

In the studied described in the previous section [49] it is assumed that upon the detection of service degradation, the system enters a digesting phase where newly arriving jobs are rejected and queued jobs are processed at a decreased service rate. The present study differs from [49] in the following points. First, we relax the assumption that all the arriving jobs are rejected during the decision process. This modification affects the underlying model of the problem formulation such that the job processing process in the deteriorating stage is no longer a pure death process. Next, we change the assumption that the decision is made only when the system state is inspected at every interval of length τ. It is reasonable to assume that the decision can be made upon the completion of the job by user notification. Hence, in this work we focus on the decision problem where a decision can be made whenever a job execution is finished. These changes in the problem settings affect the problem definition, and hence we clarify the scope of the problem in the next section.

4.2.3.3. Problem statement

In this study, we consider a job processing system with software rejuvenation that was also explained in Section 4.2.2.2.

4.2.3.3.1. System model

We define the states of a job processing system by the number of queued jobs in the system. Denote by $E_i, i \in \mathbb{Z}_+ = \{0,1,2,\cdots\}$ the aging states with *i* jobs. When software rejuvenation is applied, the system enters the rejuvenation state *R* at which all the queued jobs are cleared. The state transition between aging states occurs either when a new job arrives or when the current job execution is completed. Job arrivals are assumed to follow a Poisson arrival process with rate λ, while the service times of the jobs are exponentially distributed with

service rate $r\mu$, where r $(0 < r < 1)$ represents the level of degradation due to software aging. The traffic intensity is defined as $\rho_r = \lambda/r\mu$ and is assumed to be less than 1. If r is sufficiently small that ρ_r is larger than 1, the system becomes unstable and the system needs to be restarted immediately. Figure 4.28 shows the state transition diagram for the deteriorating job processing system.

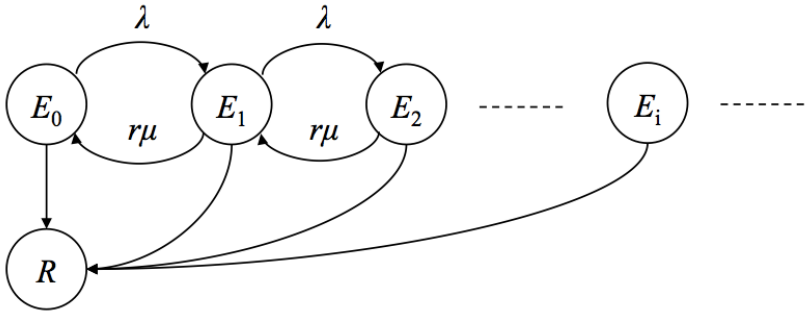


Figure 4.28 State transition diagram of a deteriorating job processing system [49]

4.2.3.3.2. Decision model

We assume that the trigger for software rejuvenation can be decided only when a job finishes its execution. With reference to the number of remaining jobs, the system can decide whether to perform software rejuvenation immediately or to continue the operation until the completion of the next job. The rejuvenation and waiting actions are denoted by a_r and a_w, respectively. When the action a_r is chosen at a decision point, all the jobs in the system are dropped, and this is regarded as the rejuvenation cost. The cost is proportional to the number of dropped jobs, and hence we define the cost for the rejuvenation action $C_{i,R}(a_r) = c_{\text{drop}} \cdot i$, where the subscript *i*,*R* represents the state transition from state E_i to the rejuvenation state *R*, and c_{drop} represents the unit cost of a dropped job (e.g., penalty specified by web application service). On the other hand, when action a_w is chosen at a decision point, the decision of rejuvenation is postponed to the next decision point when the execution of the next job finishes. The job in this period is processed under the deteriorating service rate and the completion of the execution must be delayed in comparison to the normal condition. Taking into account the delayed job execution, we define the cost for the waiting action $C_{i,j}(a_w) = c_{\text{delay}}$, where the subscript *i*,*j* represents the state transition from state E_i to state

$E_j, j \geq i-1$, and c_{delay} represents the unit cost of a delayed job (e.g., penalty on service performance in service provider). Note that the next state E_j depends on the number of observed job arrivals by the completion of the current job execution. We only consider the case where c_{delay} is smaller than c_{drop} because otherwise the decision problem becomes trivial as a_r is the best option in any state.

4.2.3.3.3. Problem formulation

The decision process for software rejuvenation terminates with the decision a_r, and hence we formulate the problem as an optimal stopping problem. Based on the principle of optimality [24], the optimality equation of the problem is given by

$$v(i) = \begin{cases} \min\left[C_{i,R}(a_r), \sum_{j=i-1}^{\infty} P_{i,j}(a_w)\left\{C_{i,j}(a_w) + v(j)\right\}\right], & i > 0 \\ 0, & i = 0 \end{cases} \quad (6)$$

where $P_{i,j}(a_w)$ represents the transition probability from the state E_i to the state E_j and $v(i)$ is called a value function that represents the value of being in the state E_i. The optimal policy that assigns the optimal actions to be taken at individual states to minimize the expected cost is given by the solution of (6).

4.2.3.4. Analysis of optimal policy

To analyze the optimal decision policy, in this section first we derive the transition probabilities $P_{i,j}(a_w)$ between decision points and then the cost for the waiting action at state E_j is elaborated. The following section is devoted to the proof of the optimal policy through two propositions.

4.2.3.4.1. Transition probability

To analyze the optimal policy for (6), the transition probability $P_{i,j}(a_w)$ needs to be elucidated. The state transition from E_i to E_j occurs when $j-i+1$ jobs arrive by the current job finishes the execution. Since the arrivals of jobs follows a Poisson process with

rate λ and the job execution time is exponentially distributed with rate $r\mu$, the joint probability density function for this transition is given by

$$f(t) = \frac{e^{-\lambda t}(\lambda t)^{j-i+1}}{(j-i+1)!} \cdot r\mu e^{-r\mu t}.$$

By integrating *t* between 0 and infinity, we obtain

$$P_{i,j}(a_w) = \int_0^\infty f(t)dt = \frac{r\mu \cdot \lambda^{j-i+1}}{(j-i+1)!} \int_0^\infty t^{j-i+1} \cdot e^{-(\lambda+r\mu)t} dt.$$

Substituting $(\lambda + r\mu)t$ into *z*,

$$\begin{aligned} P_{i,j}(a_w) &= \frac{r\mu \cdot \lambda^{j-i+1}}{(j-i+1)! \cdot (\lambda+r\mu)^{j-i+2}} \int_0^\infty z^{j-i+1} \cdot e^{-z} dz \\ &= \frac{r\mu \cdot \lambda^{j-i+1} \cdot (j-i+1)!}{(j-i+1)! \cdot (\lambda+r\mu)^{j-i+2}} \\ &= \frac{{\rho_r}^{j-i+1}}{(1+\rho_r)^{j-i+2}}. \end{aligned}$$

Therefore, the number of arrival jobs $k = j - i + 1$ follows a modified geometric distribution with parameter $1/(1+\rho_r)$ [70].

4.2.3.4.2. Expected cost for wait action

Given an admissible policy for optimal stopping, the expected cost for choosing the action a_w at E_i can be expressed by $\varphi(i)$ as below:

$$\begin{aligned} \varphi(i) &= \sum_{j=i-1}^{\infty} P_{i,j}(a_w) \left\{ C_{i,j}(a_w) + v(j) \right\} \\ &= c_{\text{delay}} + \sum_{j=i-1}^{\infty} \frac{{\rho_r}^{j-i+1}}{(1+\rho_r)^{j-i+2}} \cdot v(j) \\ &= c_{\text{delay}} + \frac{1}{1+\rho_r} \sum_{k=0}^{\infty} \left(\frac{\rho_r}{1+\rho_r} \right)^k \cdot v(i+k-1). \end{aligned}$$

By definition, $\varphi(i)$ is the expected cost when action a_w is chosen at E_i, indicating $\varphi(i) \leq C_{i,R}(a_r)$. Thus, replacing $v(i)$ in the right-hand side with $\varphi(i)$ and transposing it to the left-hand side, we obtain

$$\varphi(i) = c_{\text{delay}} + \frac{1}{1+\rho_r}\left[v(i-1) + \frac{\rho_r}{1+\rho_r}\cdot v(i) + \sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot v(i+k-1)\right]$$

$$= \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\left[(1+\rho_r)\cdot c_{\text{delay}} + v(i-1) + \sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot v(i+k-1)\right]. \quad (7)$$

When $\varphi(i)$ is equal to $C_{i,R}(a_r)$, the expected cost for action a_w is the same as the cost for rejuvenation. We assume that action a_r takes priority over action a_w in this case.

4.2.3.4.3. Optimal policy

The optimal policy determines the action to be taken at each state such that the total cost until rejuvenation is minimized. In order to analytically derive the optimal policy for rejuvenation, we have to explore the policies that can satisfy the optimality equation (6). A major obstacle to finding the solution of the optimality equation is the interrelations among the values of the states where the number of states is infinite. As can be seen in expression (7), the cost of waiting action $\varphi(i)$ depends on all the values of $v(i-1)$ and $v(i+k-1), k \geq 2$.

To overcome this, we start by considering the action chosen at E_i as *i* is very large. We consider the action decisions at states with a very large number of queued jobs. The action can be a_r or a_w by the problem definition. If the action a_r is chosen at all states E_i $(i > i^*)$, i^* can be the maximum value where the action a_w is chosen. Section 4.3.1 considers this case and shows that the action a_w cannot be chosen for state $E_i, i > 0$. On the other hand, Section 4.3.2 deals with the opposite case where the action a_w is always chosen at any state $E_i, i > i^*$.

<u>*Rejuvenation case:*</u>

Here, we show the condition for which the action a_r is always chosen under the optimal policy.

Proposition 4.2.3.1

The action a_r is chosen at every state $E_i, i \geq 0$, if and only if $c_{\text{delay}} \geq (1-\rho_r) \cdot c_{\text{drop}}$.

To prove Proposition 4.2.3.1, the following two lemmas are required.

Lemma 4.2.3.1

For any integer values *i* larger than $i^* \geq 1$, when the action a_r is chosen at every state E_i, the costs c_{drop} and c_{delay} satisfy the condition $c_{\text{delay}} \geq (1-\rho_r) \cdot c_{\text{drop}}$.

Lemma 4.2.3.2

For any integer values *i* larger than $i^* \geq 1$, when the action a_r is chosen at every state E_i and the action a_w is chosen at state E_{i^*}, action a_w should be chosen at every state $E_l, 0 < l \leq i^*$.

In the proof of Proposition 4.2.3.1, we show that there does not exist $i^* \geq 1$ satisfying Lemma 4.2.3.2. As a consequence, we show that the action a_r must be chosen at every state $E_i, i \geq 0$. From this result, the precondition of Lemma 4.2.3.1, the action a_r is chosen at every state E_i, is satisfied. This proves the if and only if statement presented in Proposition 4.2.3.1. The formal proofs of the lemmas and the proposition are given below.

Proof of Lemma 4.2.3.1

Assuming action a_r is always chosen at any state $E_i, i > i^*$, we show that $c_{\text{delay}} \geq (1-\rho_r) \cdot c_{\text{drop}}$ must be satisfied. From the precondition, we have $v(i) = C_{i,R}(a_r) = c_{\text{drop}} \cdot i$ for $i > i^*$. From expression (7), the expected cost for the wait action at E_i is represented by

$$\varphi(i) = \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\left[(1+\rho_r)\cdot c_{\text{delay}} + v(i-1) + \sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot v(i+k-1)\right]$$

$$= \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\left[(1+\rho_r)\cdot c_{delay} + v(i-1) + \sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot c_{drop} \cdot (i+k-1)\right]. \quad (8)$$

The infinite series in the last term can be expanded and rearranged as below.

$$\sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k\cdot(i+k-1)=\sum_{k=1}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k\cdot(i+k-1)-\left(\frac{\rho_r}{1+\rho_r}\right)\cdot i$$

$$=\sum_{k=1}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k\cdot(i-1)+\sum_{k=1}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k\cdot k-\left(\frac{\rho_r}{1+\rho_r}\right)\cdot i$$

$$=\rho_r\cdot(i-1)+(1+\rho_r)\cdot\rho_r-\left(\frac{\rho_r}{1+\rho_r}\right)\cdot i$$

$$=\rho_r{}^2+\frac{\rho_r{}^2}{1+\rho_r}\cdot i. \tag{9}$$

Substituting the last term of (8) with (9), and moving $i\cdot c_{\text{drop}}$ from the right to the left, we have

$$\varphi(i)-i\cdot c_{\text{drop}}=\frac{1+\rho_r}{1+\rho_r+\rho_r{}^2}\left[(1+\rho_r)\cdot c_{\text{delay}}+v(i-1)+(\rho_r{}^2-i)\cdot c_{\text{drop}}\right]$$

$$=\frac{1+\rho_r}{1+\rho_r+\rho_r{}^2}\left[(1+\rho_r)\{c_{\text{delay}}-(1-\rho_r)c_{\text{drop}}\}+v(i-1)-(i-1)c_{\text{drop}}\right]. \tag{10}$$

Since action a_r is chosen at the state E_i, it requires $\varphi(i)-i\cdot c_{\text{drop}}\geq 0$ for any i. For $i>i^*+1$, $v(i-1)$ is equal to $(i-1)\cdot c_{\text{drop}}$ because action a_r is chosen. Therefore, to ensure the right-hand side of expression (10) is positive, the condition $c_{\text{delay}}-(1-\rho_r)\cdot c_{\text{drop}}\geq 0$ must be satisfied. ■

Proof of Lemma 2

First, we show that action a_w is chosen at E_{i^*-1} under the conditions given in Lemma 4.2.3.2 and then we show that action a_w is also chosen at the states $E_l, 0<l\leq i^*$, by induction. The expected cost for the wait action at E_{i^*} can be derived in a similar manner to the proof of Lemma 4.2.3.1,

$$\varphi(i^*)-i^*\cdot c_{\text{drop}}$$

$$=\frac{1+\rho_r}{1+\rho_r+\rho_r{}^2}\left[(1+\rho_r)\cdot\{c_{\text{delay}}-(1-\rho_r)\cdot c_{\text{drop}}\}+v(i^*-1)\right. \tag{11}$$

$$\left.-(i^*-1)\cdot c_{\text{drop}}\right].$$

Since action a_w is chosen at state E_{i^*}, $\varphi(i^*)-i^*\cdot c_{\text{drop}}<0$ and hence

$$v(i^*-1)-(i^*-1)\cdot c_{\text{drop}} < -(1+\rho_r)\cdot\{c_{\text{delay}}-(1-\rho_r)\cdot c_{\text{drop}}\}. \quad (12)$$

The right-hand side of expression (12) is negative because $(1-\rho_r)\cdot c_{\text{drop}} \le c_{\text{delay}}$ by Lemma 4.2.3.1. Thereby, $v(i^*-1) < (i^*-1)\cdot c_{\text{drop}}$, which means action a_w is also chosen at state E_{i^*-1}. Since, expression (11) is satisfied only for i^*, we need to further investigate the cases of $E_l, 0 < l < i^*$. Starting from the given condition $\varphi(i^*) - i^* \cdot c_{\text{drop}} < 0$, to determine the action chosen for E_l, we rely on proof basically by induction.

For simplicity, we introduce the following notations.

$$\mathcal{D}_i \equiv \varphi(i) - i\cdot c_{\text{drop}},$$

$$\mathcal{C} \equiv (1+\rho_r)\cdot\{c_{\text{delay}}-(1-\rho_r)\cdot c_{\text{drop}}\}.$$

Note that $\mathcal{D}_i$ indicates the decision at state E_i where action a_w is chosen if $\mathcal{D}_i < 0$, otherwise action a_r is chosen. With the above notations, expression (12) can be rewritten as $\mathcal{D}_{i^*-1} < -\mathcal{C}$. The remainder of the proof is devoted to showing $\mathcal{D}_{i^*-m} < 0$, for $m \ge 2$ given $\mathcal{D}_{i^*-m+1} < 0$.

Consider the action at state $E_{i^*-m}, m \ge 2$, given that action a_w is chosen at $E_l, i^* \ge l \ge i^*-m+1$. The expected cost for action a_w at E_{i^*-m+1} can be derived from (7),

$$\begin{aligned}
&\varphi(i^*-m+1) \\
&= \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\Bigg[(1+\rho_r)\cdot c_{\text{delay}} + v(i^*-m) \\
&\quad + \sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot v(i^*+k-m)\Bigg] \\
&= \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\Bigg[(1+\rho_r)\cdot c_{\text{delay}} + v(i^*-m) + \sum_{k=2}^{m}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot \varphi(i^*+k-m) \\
&\quad + \sum_{k=1}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot (i^*+k-m)\cdot c_{\text{drop}} \\
&\quad - \sum_{k=1}^{m}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot (i^*+k-m)\cdot c_{\text{drop}}\Bigg]
\end{aligned}$$

$$= \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\Bigg[(1+\rho_r)\cdot c_{\text{delay}} + v(i^*-m) + \sum_{k=2}^{m}\left(\frac{\rho_r}{1+\rho_r}\right)^k \{\varphi(i^*+k-m) - (i^*+k-m)\cdot c_{\text{drop}}\} - \frac{\rho_r}{1+\rho_r} \cdot (i^*+1-m)\cdot c_{\text{drop}} + \sum_{k=1}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot (i^*+k-m)\cdot c_{\text{drop}}\Bigg]$$

$$= \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\Bigg[(1+\rho_r)\cdot c_{\text{delay}} + v(i^*-m) + \sum_{k=2}^{m}\left(\frac{\rho_r}{1+\rho_r}\right)^k \{\varphi(i^*+k-m) - (i^*+k-m)\cdot c_{\text{drop}}\} + \frac{{\rho_r}^2}{1+\rho_r} \cdot (i^*+1-m)\cdot c_{\text{drop}} + {\rho_r}^2 \cdot c_{\text{drop}}\Bigg].$$

Subtracting $(i^*-m+1)\cdot c_{\text{drop}}$ from both sides,

$$\varphi(i^*-m+1) - (i^*-m+1)\cdot c_{\text{drop}} = \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\Bigg[(1+\rho_r)\cdot\{c_{\text{delay}} - (1-\rho_r)\cdot c_{\text{drop}}\} + v(i^*-m) - (i^*-m)\cdot c_{\text{drop}} + \sum_{k=2}^{m}\left(\frac{\rho_r}{1+\rho_r}\right)^k \{\varphi(i^*+k-m) - (i^*+k-m)\cdot c_{\text{drop}}\}\Bigg].$$

Using the new notations $\mathcal{D}_i$ and $\mathcal{C}$, we can rewrite the expression to

$$v(i^*-m) - (i^*-m)\cdot c_{\text{drop}} = \frac{1+\rho_r+{\rho_r}^2}{1+\rho_r}\cdot \mathcal{D}_{i^*-m+1} - \mathcal{C} - \sum_{k=2}^{m}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot \mathcal{D}_{i^*+k-m}. \quad (13)$$

When $m = 2$,

$$v(i^*-2) - (i^*-2)\cdot c_{\text{drop}} = \frac{1+\rho_r+{\rho_r}^2}{1+\rho_r}\cdot \mathcal{D}_{i^*-1} - \mathcal{C} - \sum_{k=2}^{2}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot \mathcal{D}_{i^*+k-2}$$

$$= \frac{1+\rho_r+{\rho_r}^2}{1+\rho_r} \cdot \mathcal{D}_{i^*-1} - \mathcal{C} - \left(\frac{\rho_r}{1+\rho_r}\right)^2 \cdot \mathcal{D}_{i^*}$$

$$= \frac{1+\rho_r+{\rho_r}^2}{1+\rho_r} \cdot \mathcal{D}_{i^*-1} - \mathcal{C} - \left(\frac{\rho_r}{1+\rho_r}\right)^2 \cdot \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2} \cdot (\mathcal{C} + \mathcal{D}_{i^*-1}),$$

substituting the expression from (11) to give the last transformation. Using the relationship $\mathcal{D}_{i^*-1} < -\mathcal{C}$, we have

$$v(i^*-2) - (i^*-2) \cdot c_{\text{drop}} < -\frac{2+2\rho_r+{\rho_r}^2}{1+\rho_r} \cdot \mathcal{C}.$$

Since the left-hand side of the above expression is strictly negative, $v(i^*-2)$ is equal to $\varphi(i^*-2)$, which is less than $(i^*-2) \cdot c_{\text{drop}}$, and hence action a_w is chosen at E_2 as well.

Next, for $m > 2$, $\mathcal{D}_{i^*-m+1}$ can be derived in a similar way

$$\mathcal{D}_{i^*-m+1} = \frac{1+\rho_r+{\rho_r}^2}{1+\rho_r} \cdot \mathcal{D}_{i^*-m+2} - \mathcal{C} - \sum_{k=2}^{m-1} \left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot \mathcal{D}_{i^*+k-m+1}$$

$$= \frac{1+\rho_r+{\rho_r}^2}{1+\rho_r} \cdot \mathcal{D}_{i^*-m+2} - \mathcal{C}$$

$$- \frac{1+\rho_r}{\rho_r} \left\{ \sum_{k'=2}^{m} \left(\frac{\rho_r}{1+\rho_r}\right)^{k'} \cdot \mathcal{D}_{i^*+k'-m} - \left(\frac{\rho_r}{1+\rho_r}\right)^2 \cdot \mathcal{D}_{i^*-m+2} \right\}$$

$$= (1+\rho_r) \cdot \mathcal{D}_{i^*-m+2} - \mathcal{C} - \frac{1+\rho_r}{\rho_r} \cdot \sum_{k'=2}^{m} \left(\frac{\rho_r}{1+\rho_r}\right)^{k'} \cdot \mathcal{D}_{i^*+k'-m}. \qquad (14)$$

Multiplying both sides of (14) by $\rho_r/(1+\rho_r)$ and subtracting them from (13), we have the following relation.

$$v(i^*-m) - (i^*-m) \cdot c_{\text{drop}} = (1+\rho_r) \cdot \mathcal{D}_{i^*-m+1} - \rho_r \cdot \mathcal{D}_{i^*-m+2} - \frac{1}{1+\rho_r} \cdot \mathcal{C}.$$

Rearranging the above relation, we obtain a geometric sequence as follows

$$\mathcal{D}_{i^*-m+1} - \{v(i^*-m) - (i^*-m) \cdot c_{\text{drop}}\} - \frac{\mathcal{C}}{1-{\rho_r}^2}$$

$$= \rho_r \cdot \left(\mathcal{D}_{i^*-m+2} - \mathcal{D}_{i^*-m+1} - \frac{\mathcal{C}}{1-{\rho_r}^2}\right)$$

$$= {\rho_r}^{m-1} \cdot \left(\mathcal{D}_{i^*} - \mathcal{D}_{i^*-1} - \frac{\mathcal{C}}{1-{\rho_r}^2}\right).$$

As a result,

$$v(i^* - m) - (i^* - m) \cdot c_{\text{drop}} = \mathcal{D}_{i^*-m+1} - \frac{\mathcal{C}}{1-{\rho_r}^2} - {\rho_r}^{m-1} \cdot \left(\mathcal{D}_{i^*} - \mathcal{D}_{i^*-1} - \frac{\mathcal{C}}{1-{\rho_r}^2}\right).$$

From expression (11), the difference between $\mathcal{D}_{i^*}$ and $\mathcal{D}_{i^*-1}$ is bounded by

$$\mathcal{D}_{i^*} - \mathcal{D}_{i^*-1} = \frac{1}{1+\rho_r+{\rho_r}^2} \cdot [(1+\rho_r) \cdot \mathcal{C} - {\rho_r}^2 \cdot \mathcal{D}_{i^*-1}] > \mathcal{C}.$$

Accordingly, $v(i^* - m) - (i^* - m) \cdot c_{\text{drop}}$ is bounded by

$$v(i^* - m) - (i^* - m) \cdot c_{\text{drop}} < \mathcal{D}_{i^*-m+1} - \frac{1-{\rho_r}^{m+1}}{1-\rho_r} \cdot \frac{\mathcal{C}}{1+\rho_r}$$

$$= \mathcal{D}_{i^*-m+1} - \frac{\mathcal{C}}{1+\rho_r} \cdot \sum_{k=0}^{m} {\rho_r}^k.$$

This indicates that $v(i^* - m) = \varphi(i^* - m)$ since $v(i^* - m) < (i^* - m) \cdot c_{\text{drop}}$ under the assumption $\mathcal{D}_{i^*-m+1} < 0$. By induction we conclude that action a_w must be chosen at states $E_l, 0 < l \leq i^*$ if action a_w is chosen at E_{i^*}. ■

Proof of Proposition 4.2.3.1

Under the condition given in Lemma 4.2.3.1, we prove by contradiction that there does not exist an $i^* > 0$ which satisfies Lemma 4.2.3.2. Assume that there exists $i^* > 0$ such that the action a_w is chosen, while action a_r is chosen at any state $E_i, i > i^*$. By Lemma 4.2.3.2, the action a_w is chosen at all states $E_l, 0 < l \leq i^*$ and $\mathcal{D}_{l-1}$ is bounded by

$$\mathcal{D}_{l-1} < \mathcal{D}_l - \frac{\mathcal{C}}{1+\rho_r} \cdot \sum_{k=0}^{i^*-l+1} {\rho_r}^k.$$

However, the above condition is not satisfied at $l = 1$ since $\mathcal{D}_0 = v(0) - 0 \cdot c_{\text{drop}} = 0$. Therefore, a nonnegative i^* does not exit. Action a_r is chosen at any states $E_i, i \geq 0$, provided that $c_{\text{delay}} \geq (1-\rho_r) \cdot c_{\text{drop}}$. Meanwhile, as proved in Lemma 4.2.3.1, $c_{\text{delay}} \geq (1-\rho_r) \cdot c_{\text{drop}}$ must be satisfied if action a_r is chosen at any states $E_i, i \geq 0$. This concludes our proof. ■

Waiting case:

In contrast to the previous case, we consider the situation where action a_w is always chosen at E_i when the number of queued jobs is larger than a certain value i^*. We show the necessary and sufficient condition for this case.

Proposition 4.2.3.2

Action a_w is chosen at every state $E_i, i \geq 0$, if and only if $c_{\text{delay}} < (1 - \rho_r) \cdot c_{\text{drop}}$.

To prove Proposition 4.2.3.2, we require the following two lemmas.

Lemma 4.2.3.3

For any integer values *i* larger than $i^* \geq 1$, given action a_w is chosen at every state E_i, the expected cost for action a_w at E_i has the following properties.

i) $\varphi(i+1) = \varphi(i) + \frac{1}{1-\rho_r} \cdot c_{\text{delay}}$.

ii) $\sum_{k=2}^{\infty} \left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot v(i+k-1) = \frac{{\rho_r}^2}{1-\rho_r} \cdot c_{\text{delay}} + \frac{{\rho_r}^2}{1+\rho_r} \cdot \varphi(i)$.

Lemma 4.2.3.4

If action a_w is chosen at every state E_i for $i > i^* \geq 1$ and action a_r is chosen at E_{i^*}, the cost parameters should satisfy the condition $c_{\text{delay}} < (1 - \rho_r) \cdot c_{\text{drop}}$.

In the proof of Proposition 4.2.3.2, we show that there does not exist $i^* \geq 1$ such that action a_r is chosen at E_{i^*} while action a_w is chosen at every state E_i for $i > i^*$. This proves that action a_w must be chosen at every state $E_i, i \geq 0$. From this, the precondition of Lemma 4.2.3.4, the action a_r is chosen at every state E_i, is satisfied. Therefore, we can prove the condition given in Proposition 4.2.3.2. The proofs of the lemmas and the proposition are given below.

Proof of Lemma 4.2.3.3

To obtain the relation given in i), we will derive the following second-order difference equation whose solution gives i).

$$\varphi(i+2) = \frac{1}{\rho_r} \cdot \left[(1+\rho_r) \cdot \varphi(i+1) - \varphi(i) - c_{\text{delay}}\right]. \tag{15}$$

From the cost function (7), we have

$$\varphi(i+1) = \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\left[(1+\rho_r) \cdot c_{\text{delay}} + v(i) + \sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot v(i+k)\right].$$

Since $v(i+k) = \varphi(i+k)$ for any $i > i^*$ and $k \geq 0$, the above expression is rewritten as

$$\varphi(i+1) = \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\left[(1+\rho_r) \cdot c_{\text{delay}} + \varphi(i) + \sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot \varphi(i+k)\right]$$

$$= \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\left[(1+\rho_r) \cdot c_{\text{delay}} + \varphi(i) + \sum_{k'=3}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^{k'-1} \cdot \varphi(i+k'-1)\right]$$

$$= \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\left[(1+\rho_r) \cdot c_{\text{delay}} + \varphi(i) + \frac{1+\rho_r}{\rho_r} \cdot \left\{\sum_{k'=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^{k'} \cdot \varphi(i+k'-1) - \left(\frac{\rho_r}{1+\rho_r}\right)^2 \cdot \varphi(i+1)\right\}\right].$$

Transposing the term $\varphi(i+1)$ in the right-hand side to the left-hand side,

$$\varphi(i+1) = \frac{1}{1+\rho_r}\left[(1+\rho_r) \cdot c_{\text{delay}} + \varphi(i) + \frac{1+\rho_r}{\rho_r} \cdot \sum_{k'=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^{k'} \cdot \varphi(i+k'-1)\right]$$

$$= c_{\text{delay}} + \frac{1}{1+\rho_r} \cdot \varphi(i) + \frac{1}{\rho_r} \cdot \sum_{k'=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^{k'} \cdot \varphi(i+k'-1). \tag{16}$$

Therefore, for $i > i^*$ we have

$$\sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot v(i+k-1) = \rho_r \cdot \varphi(i+1) - \rho_r \cdot c_{\text{delay}} - \frac{\rho_r}{1+\rho_r} \cdot \varphi(i). \tag{17}$$

Consider the expected cost for action a_w at E_{i+2}, by (16)

$$\varphi(i+2) = c_{\text{delay}} + \frac{1}{1+\rho_r} \cdot \varphi(i+1) + \frac{1}{\rho_r} \cdot \sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot \varphi(i+k)$$

$$= c_{\text{delay}} + \frac{1}{1+\rho_r} \cdot \varphi(i+1) + \frac{1}{\rho_r} \cdot \sum_{k'=3}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^{k'-1} \cdot \varphi(i+k'-1)$$

$$= c_{\text{delay}} + \frac{1}{1+\rho_r} \cdot \varphi(i+1) + \frac{1}{\rho_r} \cdot \left(\frac{1+\rho_r}{\rho_r}\right) \cdot \left\{ \sum_{k'=2}^{\infty} \left(\frac{\rho_r}{1+\rho_r}\right)^{k'} \cdot \varphi(i+k'-1) - \left(\frac{\rho_r}{1+\rho_r}\right)^2 \cdot \varphi(i+1) \right\}$$

$$= c_{\text{delay}} + \frac{1+\rho_r}{{\rho_r}^2} \cdot \sum_{k'=2}^{\infty} \left(\frac{\rho_r}{1+\rho_r}\right)^{k'} \cdot \varphi(i+k'-1).$$

Using (17), the following second-order difference equation is obtained.

$$\varphi(i+2) = c_{\text{delay}} + \frac{1+\rho_r}{{\rho_r}^2} \cdot \left[\rho_r \cdot \varphi(i+1) - \rho_r \cdot c_{\text{delay}} - \frac{\rho_r}{1+\rho_r} \cdot \varphi(i)\right]$$

$$= \frac{1}{\rho_r} \cdot \left[(1+\rho_r) \cdot \varphi(i+1) - \varphi(i) - c_{\text{delay}}\right].$$

Solving the difference equation yields the property i). Substituting i) into equation (17), we obtain the property ii) ■

Proof of Lemma 4.2.3.4

From the precondition, action a_w is chosen at any state E_i for $i > i^* \geq 1$, and Lemma 4.2.3.3 holds for $i > i^*$. Thus, applying property i) of Lemma 4.2.3.3 to expression (7),

$$\varphi(i) = v(i-1) + \frac{1}{1-\rho_r} \cdot c_{\text{delay}}.$$

Applying $i^* + 1$ to i in the above equation,

$$\varphi(i^*+1) = v(i^*) + \frac{1}{1-\rho_r} \cdot c_{\text{delay}} = i^* \cdot c_{\text{drop}} + \frac{1}{1-\rho_r} \cdot c_{\text{delay}}.$$

Since action a_w is chosen at E_{i^*+1}, $\varphi(i^*+1) < (i^*+1) \cdot c_{\text{drop}}$ and thus c_{delay} satisfies the condition $c_{\text{delay}} < (1-\rho_r) \cdot c_{\text{drop}}$. ■

Proof of Proposition 2

We prove by contradiction that action a_w is chosen at any state $E_i, i \geq 0$, if $c_{\text{delay}} < (1-\rho_r) \cdot c_{\text{drop}}$. Suppose there exists a value $i^* \geq 1$ at which action a_r is chosen under the optimal policy, and for any $i > i^*$ action a_w is chosen at every state E_i. Consider the expected cost for a_w at E_{i^*}, from expression (7),

$$\varphi(i^*) = \frac{1+\rho_r}{1+\rho_r+{\rho_r}^2}\left[(1+\rho_r)\cdot c_{\text{delay}} + v(i^*-1) + \sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot v(i^*+k-1)\right]. \quad (18)$$

Noting that property ii) of Lemma 4.2.3.3 holds for $i > i^*$, the last term of the above expression can be rewritten as

$$\begin{aligned}
&\sum_{k=2}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^k \cdot v(i^*+k-1) \\
&= \frac{\rho_r}{1+\rho_r}\left[\frac{\rho_r}{1+\rho_r}\cdot v(i^*+1) + \sum_{k=3}^{\infty}\left(\frac{\rho_r}{1+\rho_r}\right)^{k-1}\cdot v(i^*+k-1)\right] \\
&= \frac{\rho_r}{1+\rho_r}\left[\frac{\rho_r}{1+\rho_r}\cdot \varphi(i^*+1) + \frac{{\rho_r}^2}{1-\rho_r}\cdot c_{\text{delay}} + \frac{{\rho_r}^2}{1+\rho_r}\cdot \varphi(i^*+1)\right] \\
&= \frac{\rho_r}{1+\rho_r}\left[\rho_r\cdot \varphi(i^*+1) + \frac{{\rho_r}^2}{1-\rho_r}\cdot c_{\text{delay}}\right].
\end{aligned}$$

Applying this relation to expression (18), the value function for i^*-1 is given by

$$v(i^*-1) = \frac{1+\rho_r+{\rho_r}^2}{1+\rho_r}\cdot\varphi(i^*) - \frac{1+\rho_r-{\rho_r}^2}{(1+\rho_r)(1-\rho_r)}\cdot c_{\text{delay}} - \frac{{\rho_r}^2}{1+\rho_r}\cdot\varphi(i^*+1).$$

From the assumption, $\varphi(i^*) \geq i^*\cdot c_{\text{drop}}$ and $\varphi(i^*+1) < (i^*+1)\cdot c_{\text{drop}}$,

$$\begin{aligned}
v(i^*-1) &> \frac{1+\rho_r+{\rho_r}^2}{1+\rho_r}\cdot i^*\cdot c_{\text{drop}} - \frac{1+\rho_r-{\rho_r}^2}{(1+\rho_r)(1-\rho_r)}\cdot c_{\text{delay}} - \frac{{\rho_r}^2}{1+\rho_r}\cdot(i^*+1)\cdot c_{\text{drop}} \\
&= \left(i^* - \frac{{\rho_r}^2}{1+\rho_r}\right)\cdot c_{\text{drop}} - \frac{1+\rho_r-{\rho_r}^2}{(1+\rho_r)(1-\rho_r)}\cdot c_{\text{delay}}.
\end{aligned}$$

From Lemma 4.2.3.4, we have $v(i^*-1) > (i^*-1)\cdot c_{\text{drop}}$. However, this contradicts the definition of the value function which must satisfy $v(i^*-1) \leq (i^*-1)\cdot c_{\text{drop}}$. Therefore, there does not exist $i^* \geq 1$ such that the action a_r is chosen at E_{i^*} while action a_w is chosen at every state $E_i, i > i^*$.

Meanwhile, when action a_w is chosen at any state $E_i, i \geq 0$, the condition $c_{\text{delay}} < (1-\rho_r)\cdot c_{\text{drop}}$ clearly holds by Lemma 4.2.3.4. The proposition has been proved. ■

Relationship between two policies:

From the propositions proved above, the optimal policy is determined by the relation among c_{delay}, c_{drop} and ρ_r. When $c_{\text{delay}} < (1-\rho_r)\cdot c_{\text{drop}}$, action a_w should be chosen regardless of the number of queued jobs. On the other hand, when $(1-\rho_r)\cdot c_{\text{drop}} \leq c_{\text{delay}} < c_{\text{drop}}$, action a_r should be chosen. Figure 4.29 shows the relationship among the parameters that characterizes the optimum decision policy.

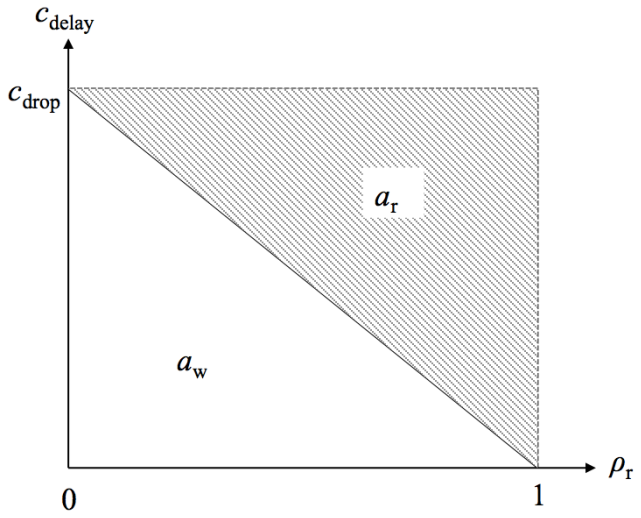


Figure 4.29 Relationship among the parameters characterizing the optimal policy [50]

The derivation of the optimal policy stands on the assumption that the degradation level of service rate is given by the constant *r*. At every decision point, it is assumed that the traffic intensity ρ_r is not changed in the future. In practical applications, however, we can update the value of ρ_r according to the latest observations to make a better decision at each decision point. In this case, software rejuvenation must not be triggered as long as the value of ρ_r is in the lower triangle of Figure 4.29, while it should be triggered when the value of ρ_r enters in the upper triangle in Figure 4.29. The derived policy can provide a simple and reasonable guide for determining the timing to trigger software rejuvenation for a deteriorating job processing system.

Threats to the validity:

The derived optimal policy provides a simple method for determining the time to software rejuvenation by simply looking at the cost balance between c_{delay} and c_{drop}, and the

deteriorating traffic intensity ρ_r. However, our analysis relies on several simplified assumptions. We revisit the assumptions we made in the derivation and consider the gap to the real systems to clarify the potential limitations of the policy and necessary enhancements.

The first and probably the most arguable assumption is the constant level of service degradation due to software aging. Once the service rate is degraded to $r\mu$, it is assumed not to change further. This assumption seems inadequate as software aging is considered to progress continuously. However, progressive software aging does not always directly link to the progressive degradation of service performance. The service performance might change drastically at once due to the accumulation of errors caused by aging-related bugs. An example of this is I/O performance degradation in operating systems due to swapping after the extended execution of software suffering from memory-leak [48]. This performance degradation is caused by the latency gap between memory access and hard disk access. Similar phenomenon can be observed in a database management system that has the limited memory caching capacity for query and table caches. Although we do not fully consider the progressive degradation of service performance, the service degradation level is assumed not to change significantly at least until the next decision point. If we can predict the future degradation of service performance, an additional uncertainty could be incorporated into the decision model as studied in [80][82].

The second simplified assumption is that the unit costs for delayed jobs and dropped jobs are deterministic and identical for all the affected jobs. Depending on the service or business offered by the system, several interpretations can be made for the cost model. One of the applications of our assumed model is the web service system that provides the service level agreement (SLA) on the performance of requested job processing and service availability. All delayed jobs that do not meet a specific deadline or service rate in the SLA will impose a penalty to the provider. On the other hand, the unavailability of service might be evaluated by the number of completed jobs. In such a service system, the costs of delayed jobs and dropped jobs are regarded to be proportional to the number of affected jobs, since they are the actual indicator of the service level. Of course, the cost model can be extended and tailored to the requirements of organizations or individuals. For instance, the cost of delayed

jobs may depend on the length of the delay and in this case the unit cost of a delayed job is not identical for all the affected jobs. In such cases, we need to extend the model so that the waiting times of individual jobs are taken into account in the cost c_{delay}. This cost model would better fit many HPC application systems. Analysis of waiting times using queueing theory or job completion time analysis [64] is an area for future work.

The third assumption is that the job arrival rate λ and the deteriorating service rate $r\mu$ are known at each decision point. We assume that the job arrival rate is constant and can be estimated from the history of job requests in advance. However, the job arrival rate may change over time during the decision process. A variable arrival rate can be included by introducing a more general arrival model such as MAP, although this may increase the complexity of the decision model. Regardless of which arrival model is employed, we need to estimate the arrival rates from the observed arrivals. In contrast to the arrival rate, to estimate the deteriorating service rate the service performance needs to be continuously monitored in the real system. Although the original service rate μ in the initial phase can be estimated in advance, the deteriorating service rate is only observable during operation. Since only one sample at the decision time is unlikely to provide an accurate estimate of the deteriorating service rate, several samples at different times need to be collected for estimation. A variety of system monitoring tools can collect performance metrics periodically, while this inspection might introduce performance overheads depending on the sampling rate or the way of monitoring instrumentation. Such overhead also needs to be investigated by experiments in real applications. This is another avenue for future research.

4.2.3.5. Summary

In this study, we analytically derived the optimal policy for condition-based software rejuvenation in a job processing system modeled by an M/M/1 queue with a deteriorating service rate. The relation among the cost of delayed jobs, the cost of dropped jobs due to rejuvenation, and the traffic intensity is the key to determining the policy, while the number of queued jobs in the system has no impact. Conventional studies rely on numerical methods to derive the optimal decision policy, while our study explicitly shows the optimal policy for

a simpler problem by assuming an M/M/1 queue. The work presented here can be extended to more complicated cases, including different cost models for job processing systems.

4.3. Software life-extension

As briefly introduced in Section 4.1, software life-extension was proposed as a new operational countermeasure to software aging. Software life-extension does not reset the system states, but postpones a potential system failure by reducing the impacts of software aging. This section describes our studies on software life-extension that was initially presented in ISSRE2012 [48] and further extended work is published as an article of IEEE Transactions on Reliability in 2017 [83]. These studies are co-authored with Dr. Yoshiharu Maeno, Dr. Jianwen Xiang and Ms. Kumiko Tadano.

4.3.1. Software life-extension and feasibility study

This section reviews the original concept of software life-extension and its feasibility study using server virtualization [48].

4.3.1.1. Introduction to software life-extension

The concept and methods of software life-extension are introduced in this section. In order to explain the conceptual difference between software life-extension and software rejuvenation, an availability model representing the behavior of software life-extension is described.

4.3.1.1.1. Concept

The term *software life-extension* comes from a natural extension of the metaphor of software aging. Software aging represents the transient state of the software execution environment, where available resources gradually decrease due to aging-related software faults. The lifetime of software execution reaches its limit when the system depletes its resources as a result of the accumulated aging effects. Software life-extension aims at postponing such failures by impeding the progress of software aging. It is a temporal mitigation to extend the

lifetime of the execution environment, but it does not provide a radical solution to the fault causing the software aging. To extend the lifetime of software execution, supplemental resources could be assigned to the execution environment if the application can make use of them. Alternatively, software aging can be impeded by decreasing the workload, provided that the rate of aging depends on the workload. These approaches do not require any changes to the source code and are often easily applicable by means of common maintenance operations, commands, or scripts. Therefore, software life-extension is a non-intrusive countermeasure to software aging.

4.3.1.1.2. Means

There are at least two conceptual ways to implement software life-extension: dynamic resource allocation and workload control. The first approach extends the lifetime of aged software through dynamic resource allocation in which the amount of resources is increased dynamically during execution. Recent advances in virtualization technologies make such resource allocations at runtime possible. For example, Xen hypervisor provides a functionality to virtualize hardware resources and allocate them to a Virtual Machine (VM). In this approach, we need standby resources which can be allocated dynamically and may be shared with other software execution environments. The use of standby resources may incur costs, such as higher resource usage costs imposed by the cloud and/or hosting service, and unavailability of other services sharing the standby resource.

The second approach controls the workload so as to decrease the load on the aged software. This approach is limited to applications that work with a workload manager or have a load balancer in the front-end. The workload is reduced by assigning jobs to other instances or dropping job requests at the workload manager or load balancer. Software aging is often associated with the workload of the software [84][85]; therefore, aging can be impeded by reducing the workload. Although this helps to extend the lifetime of the software, resource exhaustion is inevitable as long as the software continues executing. This approach can be considered to be like designing a system that can survive even in the case of a component failure. Unlike typical degradable systems, software life-extension using workload control

does not guarantee that the software will continue to execute. Even after a life-extension, the software may eventually encounter a failure due to resource exhaustion because life-extension itself does not remove the root-cause. Similar to the first approach, workload control may also incur additional problems, such as workload reallocations that overload other instances and the workload manager rejecting requests.

Consequently, although both of these approaches are feasible in a real system they require specific system configurations, preparations and resources. The appropriate means should be decided considering the application type and system environment. In the following section, we show the feasibility of software life-extension by taking the dynamic resource allocation approach.

4.3.1.1.3. Advantages and drawbacks

Regardless of the above-mentioned means, the primary advantage of software life-extension over software rejuvenation is continuous execution even as the software ages. Although software rejuvenation clears the aging states in a relatively short amount of downtime, it interrupts the software execution and loses potentially valuable data in memory. In contrast, software life-extension can maintain availability without any interruptions as long as possible. When an application requests a job requiring a long execution time and the question is whether or not the job will complete, life-extension is preferable to rejuvenation. Software life-extension is also suitable for applications with predetermined mission times. We can use it to meet the mission time requirement when the software is likely to finish execution before the mission time is up.

Another benefit of software life-extension is its capability of preserving memory content, as mentioned in the Introduction. The persistence of data accumulated in memory is essential to some forms of software. Software rejuvenation completely erases such data, and thus, it may cause a serious degradation in service quality. A typical example of such important memory content is paging data in an operating system. The deletion of paging data during a reboot causes a performance degradation, as reported in [86]. In contrast, software life-extension attempts to preserve memory content as long as possible. While the content of

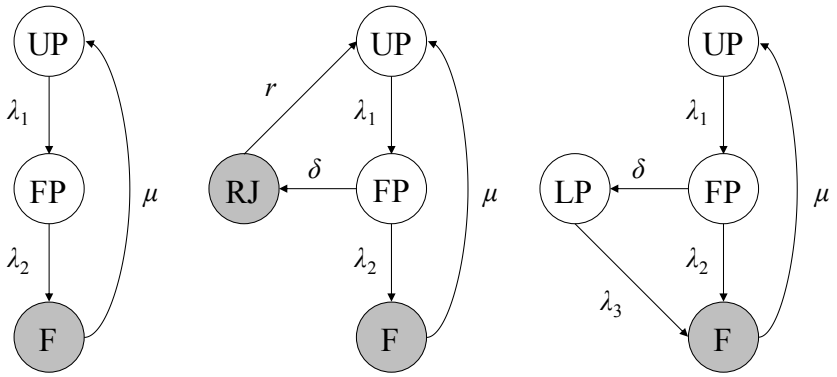
memory is eventually lost at the end of the system's life, the user may wisely use the residual lifetime to make a backup or take a snapshot and save it in persistent storage.

As discussed earlier, software life-extension incurs additional resource usage costs, performance degradations, and degradations to the availability of other services. These are potential drawbacks if they become unacceptably large or unpredictable. The trade-off is the additional lifetime in exchange for these costs. Although rejuvenation imposes an additional downtime cost, it does not require a specific system configuration (e.g., a load balancer) or any standby resources.

Unlike the hot-fix approach that corrects the source code by removing the source of software aging [87], software life-extension does not remove the source. Hence, relying on software life-extension for a long time may hinder the chances of finding and removing the root cause of the aging, which is something that system administrators should be aware of when they consider using life-extension.

4.3.1.1.4. Availability model

To characterize our approach in comparison to software rejuvenation, we introduce a continuous time Markov chain (CTMC) which represents the general behavior of software aging and software life-extension. First, Figure 4.30(a) shows a CTMC representing behavior of software aging as studied in [44]. Software starts the execution from UP state that is a highly robust state. After a certain time interval, the software proceeds to failure-probable (FP) state. Following that, software goes from FP state to failure (F) state which is represented by a shaded circle. This two-step failure behavior is commonly adopted in availability modeling and analysis of software aging [44][52]. After recovery from a failure, the software returns to UP state from F state.



(a) Software aging (b) Software rejuvenation (c) Software life-extension

Figure 4.30 CTMC models capturing the behavior of (a) software aging, (b) software rejuvenation and (c) software life-extension [48]

Let us denote λ_1, λ_2, μ as the rates attached to the transitions from UP to FP, FP to F, and F to UP, respectively. By solving the CTMC, the availability of the software A_{N} is computed as the sum of steady-state probabilities in UP state and FP state:

$$A_{\mathrm{N}} = \pi_{\mathrm{UP}} + \pi_{\mathrm{FP}} = \frac{(\lambda_1 + \lambda_2) \cdot \mu}{(\lambda_1 + \lambda_2) \cdot \mu + \lambda_1 \lambda_2}.$$

Next, the CTMC shown in Figure 4.30 (b) adds the behavior of software rejuvenation to the aging model. Software rejuvenation is performed when the state of the software is changed from FP state to RJ state which represents the rejuvenation state and is denoted by a shaded circle. When the rejuvenation completes, the software returns to UP state. Let us denote δ_r and r as the rates attached to the transitions from FP to RJ and RJ to UP, respectively. The availability of the software with software rejuvenation is computed by

$$A_{\mathrm{R}} = \frac{r\mu \cdot (\lambda_1 + \lambda_2 + \delta_r)}{r\mu \cdot (\lambda_1 + \lambda_2 + \delta_r) + \lambda_1 \cdot (\lambda_2 r + \mu \delta_r)}.$$

Then we introduce a CTMC for software life-extension as shown in Figure 4.30 (c). Instead of the rejuvenation state, we add a life-prolonging state, LP, to the aging model. The software in FP state may go to LP state by software life-extension before encountering a failure. The software will eventually fail and the state is changed from LP to F. Let us denote

δ_l and λ_3 as the rates attached to the transitions from FP to LP and LP to F, respectively. Since LP state is also a working state, the availability of the software A_{L} is computed as the sum of steady-state probabilities in UP state, FP state and LP state:

$$A_{\mathrm{L}} = \frac{\mu\lambda_3 \cdot (\lambda_1 + \lambda_2 + \delta_l) + \lambda_1\mu\delta_l}{\mu\lambda_3 \cdot (\lambda_1 + \lambda_2 + \delta_l) + \lambda_1\lambda_3 \cdot (\lambda_2 + \delta_l) + \lambda_1\mu\delta_l}.$$

The expression shows that the effects of software life-extension on the availability depend on the failure rates (λ_1, λ_2, and λ_3), the failure recovery rate (μ), and the life-extension trigger rate (δ_l). The difference between A_{N} and A_{L} is computed as

$$A_{\mathrm{L}} - A_{\mathrm{N}} = \frac{\lambda_1\mu\delta_l \cdot (\lambda_1 + \lambda_2) \cdot (\lambda_2 - \lambda_3)}{[\mu\lambda_3 \cdot (\lambda_1 + \lambda_2 + \delta_l) + \lambda_1\lambda_3 \cdot (\lambda_2 + \delta_l) + \lambda_1\mu\delta_l] \cdot [(\lambda_1 + \lambda_2) \cdot \mu + \lambda_1\lambda_2]}.$$

It indicates that the sign of A_{L}-A_{N} depends on the sign of the term ($\lambda_2 - \lambda_3$). If λ_3 is smaller than λ_2, A_{L} becomes larger than A_{N} resulting it software life-extension being effective in terms of steady-state availability. Software life-extension should be effective when it decreases the failure rate ($\lambda_3<\lambda_2$), therefore the derived condition looks reasonable.

To figure out the condition where software life-extension offers a better solution than software rejuvenation, we take the difference between A_{R} and A_{L}

$$A_{\mathrm{L}} - A_{\mathrm{R}} = \frac{\lambda_1\mu[\lambda_1\mu\delta_l\delta_r - r\lambda_3 \cdot (\lambda_1\delta_l + \lambda_2\delta_r + \delta_l\delta_r)]}{[\mu\lambda_3 \cdot (\lambda_1 + \lambda_2 + \delta_l) + \lambda_1\lambda_3 \cdot (\lambda_2 + \delta_l) + \lambda_1\mu\delta_l] \cdot [r\mu \cdot (\lambda_1 + \lambda_2 + \delta_r) + \lambda_1 \cdot (\lambda_2 r + \mu\delta_r)]}.$$

Comparing the difference with 0, from the numerator, we get the inequality

$$r\lambda_3 < \frac{\lambda_1\mu\delta_l\delta_r}{\lambda_1\delta_l + \lambda_2\delta_r + \delta_l\delta_r}$$

representing the condition where software life-extension achieves higher availability than software rejuvenation (i.e., $A_{\mathrm{L}}>A_{\mathrm{N}}$). If the value of λ_3 in the left-hand term is small enough, the above condition is likely to hold. On the other hand, if the value of r in the same term is relatively large, the condition is likely not to hold which results in A_{R} becoming larger than A_{L}.

4.3.1.2. Experimental results

This section provides an illustrative example of software life-extension on memcached. A memcached server is deployed on a VM and its lifetime is extended by software life-extension using dynamic memory allocation supported by Xen hypervisor.

4.3.1.2.1. Memcached

Memcached is an in-memory key-value store for caching objects. In large-scale web systems, memcached is widely used as a cache server for database to speed up query response. It simply implements a hash table whose content is read or inserted by corresponding keys. In spite of its simple architecture, memcached is scalable to multiple servers and achieves high-performance. Unlike the persistent database server such as MySQL, storage in memory is the heart of memcached which enables faster access to the data.

4.3.1.2.2. Problem

Memcached consumes memory for storing key-value pairs in a hash-table until it reaches the maximum limit. When the table becomes full, a subsequent insert request purges older data in least recently used (LRU) order. The limit is set to 64MB by default but it is configurable by a command option "-m". A possible problem comes from a misreading of this limit setting. This option is used to set the maximum size of memory for cache data and it does not count the usage of memory by memcached process and associated metadata. Consequently, the actual usage of memory may be increased beyond the set maximum value. If the maximum value is not set properly within the range of available free memory in consideration with additional memory consumed by memcached process, it might cause a number of swaps and further induce a crash due to out-of-memory. Users are responsible for setting the maximum limit correctly in accordance with the available resources on the execution environment, but sometimes it is hard to estimate the available memory in advance. Human configuration errors may also occur, especially in virtualized systems.

As an illustrative example of memory leak problems in memcached, let us consider a memcached deployment on a VM. A VM is configured to 1024MB of maximum memory

allocation and 512MB of initial memory allocation. The VM starts with only 512MB of memory but it can be increased up to 1024MB later. Let us suppose that a memcached is run on a VM with the "-m" option and the maximum limit is set to 1024MB by user. There is no problem in using memcached as long as the cached content does not exceed the value of 512MB that is the initial memory allocation. However, if the data inserted exceeds the value of 512MB and there is no action to increase memory allocation, memcached uses up swap spaces and the VM crashes after all due to out-of-memory.

It is not appropriate to categorize the issue as a software failure caused by aging-related bug, but the software aging can occur due to configuration mistakes by users and it will cause the accumulation of memory usage.

4.3.1.2.3. Software life-extension for memcached

A simple solution to the illustrative problem is dynamic additional memory allocation to the VM. If we can increase the free memory of the VM dynamically and memcached makes use of them, the life of memcached can be extended. We examined this on Xen hypervisor version 4.1. Figure 4.31 shows our test bed configuration.

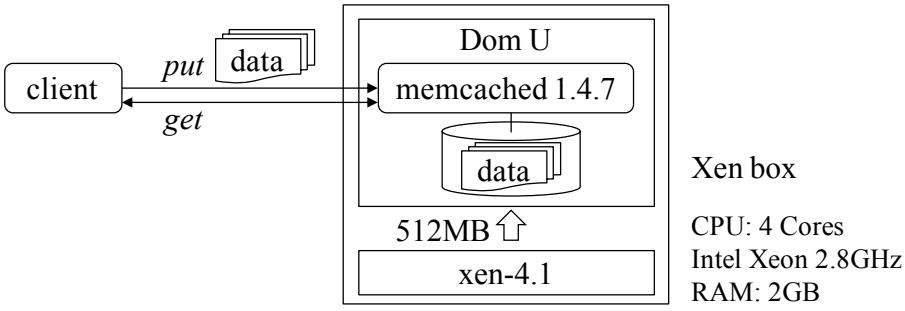


Figure 4.31 Test bed configuration [48]

A VM is created on Xen box and 512MB of memory is initially allocated to the VM. The VM has 512MB of swap space. A single instance of memcached is deployed on the VM and it starts with the "-m" option and set the maximum limit to 900MB. A client program generates requests to the memcached for load test that repeatedly inserts 1MB of unique data and reads it in a subsequent access. During the load generation, we do not insert any delays

between consecutive requests. According to the number of insert requests, the memory consumption of memcached increases gradually because the data is cached in memory. When the number of insert events exceeds 500, memory swapping is started. If no counteractions are performed, the VM is finally crashed due to out-of-memory. The changes of free memory and swap usage in the VM during this experiment are shown in Figure 4.32. The VM is crashed when it uses up both of the free memory and available swap spaces.

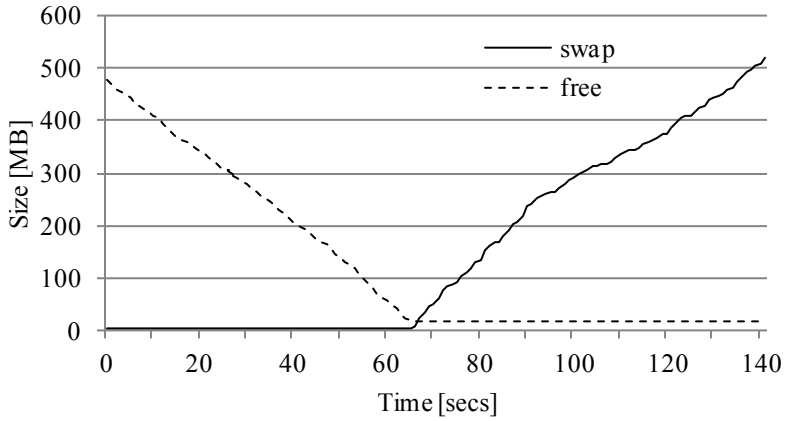


Figure 4.32 Decrease in free memory and increase in swap spaces [48]

We can postpone the failure by allocating additional memory to the VM, namely by software life-extension. When the swap usage exceeds 400MB, we allocate 88MB of additional memory to the VM through the Xen's command line interface. The total memory is increased to 600MB and the life of the VM is extended. The time to failures (TTF) observed in three experiments under the same load test are shown in Table 4.4.

Table 4.4 Time to failures with/without software life-extension [48]

Test No.	w/o life-extension	w/ life-extension
1	143	1953
2	124	2042
3	134	1635
Average	134	1877

The software life-extension prolongs the lifetime of the VM more than ten times longer than the original lifetime. Although the amount of added memory is relatively small, the lifetime is greatly improved.

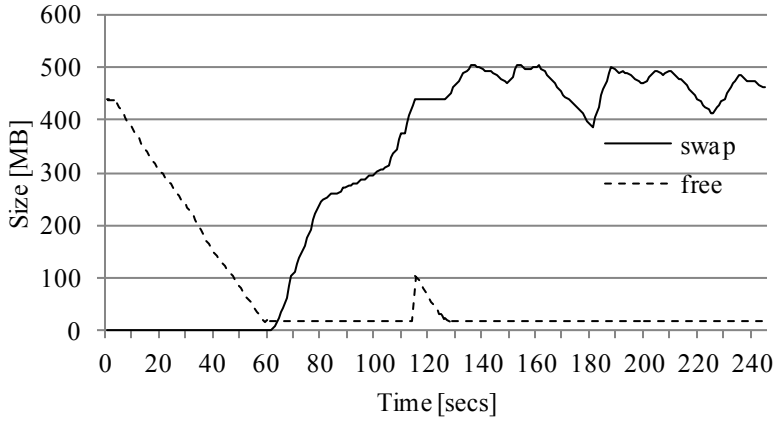


Figure 4.33 Changes in free memory and swap usage by software life-extension [48]

The amount of free memory and swap usage are changed by software life-extension as shown in Figure 4.33. The steep increase in the swap usage stops at the time when software life-extension is performed but it increases again in a short period of time after using out the added free memory. The swap usage reaches 500MB around 135 seconds. The VM does not immediately fall into out-of-memory but the swap usage fluctuates around 500MB with a subtle upward trend.

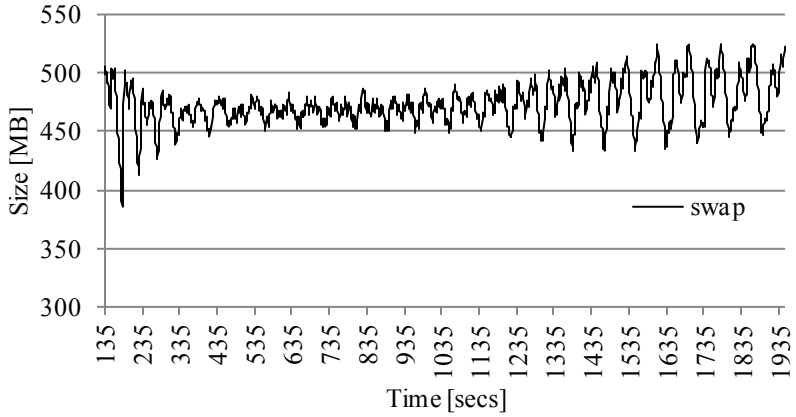


Figure 4.34 Increase in swap usage after software life-extension [48]

Figure 4.34 shows the observed swap usages from the time of 135 seconds to the time when the VM fails. The upward trend is confirmed by the normal approximation test using Mann-Kendall statistic [70] with 99% confidence level and Sen's slope estimate [88] gives the estimated trend as 13.816 KB/s.

When we allocate 600MB of RAM at the beginning rather than adding later during the runtime, such life-extension effects are not always demonstrated although the VM sometimes survives more than 2000 seconds. The preliminary result implies that the combination of the amount of the initial memory allocation and the timing to allocate the additional memory is a key to extend the lifetime. We further investigate the lifetimes with different memory configurations in the following sections.

4.3.1.2.4. Observations

Let denote M_{max} as the maximum limit (MB) set by the "-m" option of memcached. The amount of available resources in the VM is the sum of the initially allocated memory M_{RAM}, dynamically allocated memory δ_{RAM} and the swap space M_{swap}. When $M_{RAM\text{-}max}$ denotes the maximum size of the VM, the sum of M_{RAM} and δ_{RAM} remains within the bound of $M_{RAM\text{-}max}$

$$M_{RAM} + \delta_{RAM} \leq M_{RAM-max}.$$

Since memcached consumes the memory up to M_{max} for data cache, the VM is tolerant to out-of-memory problem as long as the following condition holds

$$M_{max} + \alpha + \beta < M_{RAM} + M_{swap} + \delta_{RAM}$$

where α is the amount of memory consumption by memcached process except for the data area and β is the amount of memory consumption by other processes on the VM. The values of α and β are varied during the lifetime while $M_{RAM,}$ M_{swap} and M_{max} are fixed values. We can control the lifetime of the VM by determining the value of δ_{RAM}.

Table 4.5 TTFs (secs) by different combinations of M_{RAM} and $M_{max,}$ where M_{swap} is 512MB and no software life-extension [48]

		M_{max} (MB)			
		700	800	900	1000
M_{RAM} (MB)	384	*	126 s	126 s	124 s
	512	No	*	135 s	142 s
	640	No	No	No	152 s

*) VM does not encounter a failure, but some requests from clients are dropped

Table 4.5 shows the observed TTFs in seconds with the different combinations of M_{RAM} and M_{max} where M_{swap} is fixed to 512MB and no software life-extension ($\delta_{RAM} = 0$). With all the values of M_{max}, TTF becomes longer as M_{RAM} increases. When the sum of M_{RAM} and M_{swap}(=512MB) rises above a certain threshold (around M_{max}+200MB), any failures of the VM are not observed in the hours of load tests. For instance, no failure is observed in the combination of [M_{RAM}=640MB, M_{max}=900MB]. There are no request errors in client side in this case. However, as denoted by asterisk in Table 4.5, in the combinations of [M_{RAM}=512MB, M_{max}=800MB] and [M_{RAM}=384MB, M_{max}=700MB], some requests errors are observed in the client side. These errors may be caused by busy states of memcached suffering from swapping. These configurations might encounter failures after long time execution, although they survived during the hours of tests.

Table 4.6 TTFs by varying δ_{RAM} where software life-extension is performed at 500MB of swaps [48]

δ_{RAM} (MB)	32	48	64	80	96
TTF (sec)	130	127	1004	1678	1858

Next, we fix M_{RAM} and M_{max} to 512MB and 900MB, respectively, and apply software life-extension by using various values of δ_{RAM}. The observed TTFs are shown in Table 4.6. In the first set of experiments, software life-extension is performed when the size of swap usage

increases over 500MB. Software life-extension does not affect the TTF until δ_{RAM} reaches 64MB but when over 64MB, it prolongs the TTF according to the amount of memory allocation.

Table 4.7 TTFs by varying δ_{RAM} where software life-extension is performed at 300MB of swaps [48]

δ_{RAM} (MB)	32	48	64	80	96
TTF (sec)	143	137	143	140	135

The results are varied if the timing of software life-extension is changed. Table 4.7 shows the results of TTFs where software life-extension is performed when the size of swap usage rises above 300MB. Software life-extension does not increase the lifetime in this case. The difference is most probably caused by the behavior of memory management function in operating system which is dynamically changed especially when the system suffers from the lack of resources. From this observation, the timing of software life-extension should be set to just before resource depletion, at least in our test bed.

4.3.1.3. Availability experiments

In order to analyze the impacts of software life-extension on system availability and performance, we conduct further experiments on memcached in the test bed. Based on the measurements, a Semi-Markov process (SMP) is introduced to model the behavior of software life-extension and to estimate the availability and performance measures.

4.3.1.3.1. Test configuration

We use the same test bed as described in Section 4.3.1.2.3. A VM is created with 512MB of memory and memcached is launched with the maximum limit set to 900MB. The VM may crash due to out-of-memory with this configuration if the number of inserts exceeds certain limit and no additional memory is allocated to the VM. We implement a script which applies software life-extension automatically when the VM is about to run out of free swap space. The script runs on a host server and monitors the free swap space of the VM every ten seconds

using Simple Network Management Protocol (SNMP). When the amount of free swap space becomes smaller than or equal to 1 MB, the script invokes Xen's command to allocate 64MB of additional memory to the VM. If the command is carried out successfully, the lifetime of the VM is prolonged. Regardless of whether life-extension is applied or not, the VM eventually fails due to out-of-memory. The failed VM is detected manually and a recovery operation is carried out accordingly. The recovery operation includes the destruction of failed VM, creation of a new VM with the same VM image, and start of memcached and some monitoring scripts.

4.3.1.3.2. Clients

Two client programs for memcached are developed for the experiments. A client program, *measurement client*, is aimed to measure the availability of memcached, while the other client, *load client*, is used for accelerating the aging by imposing numerous insert operations. The load client requests a "put" operation which inserts 1MB of data to the memcached by a key that is randomly sampled from the set S_{key} which contains 10000 unique keys. The requests are generated in the way that the memcached sees Poisson arrival with rate λ_{load}=10 [1/sec]. On the other hand, the measurement client requests a "get" operation which looks up the cached data in the memcached by a specific key that is sampled from the subset $S'_{key} \subset S_{key}$ which contains 1000 unique keys. As $|S'_{key}|$ is one tenth of $|S_{key}|$, we assume here the locality of cache access. With a certain probability, the request encounters a cache miss upon which the client subsequently requests a "put" operation to insert 1MB of data. The time between requests is sampled from exponential distribution with rate λ_{mes}=1 [1/sec]. The measurement client executes continuously even when the memcached is not available so that it counts the number of request drops during VM failure.

4.3.1.3.3. Measurement results

We carried out repeated life time tests in each of which a memcached starts on a VM, the VM fails due to out-of-memory, and a new VM is created for the next test. The lifetime events such as memcached startup, start swapping, VM failure, software life-extension are recorded

in log files and the time to start swap (TTS), the time to VM failure (TTF), the time to apply software life-extension (TTL), and the time spent for manual VM recovery (TTR) are computed from the log. The observed VM lifetimes by ten times of tests are shown in Table 4.8.

Table 4.8 Observed VM lifetimes by ten times of life tests [48]

Test No.	Life extension	TTS [sec]	TTF [sec]	TTL [sec]	TTR [sec]
1	Applied	71	1550	348	189
2	Not applied	70	329		169
3	Not applied	74	326		230
4	Not applied	75	327		219
5	Applied	76	1440	318	201
6	Applied	72	1518	330	211
7	Applied	81	1512	347	180
8	Applied	72	1494	341	177
9	Applied	79	1568	339	177
10	Not applied	72	363		217

Software life-extension is applied successfully six times in the experiments and it extends the lifetime of the VM more than four times when life-extension is applied. In these experiments, the success probability of software life-extension mainly depends on the coverage of detection of critical states. If we decrease the monitoring interval to observe the free swap space more frequently, the coverage might be increased. The detection coverage, in this experimental study, is roughly estimated 60% from the empirical data. Let TTF_i be the TTF observed in the life test i and TTR_i be the TTR in the life test i, respectively. The empirical availability of the VM is computed as:

$$A_{\mathrm{L}} = \frac{\sum_{i=1}^{n} TTF_i}{\sum_{i=1}^{n} (TTF_i + TTR_i)} = 0.841091.$$

The availability without software life-extension can be estimated by counting the test results for which software life-extension is not applied:

$$A_{\mathrm{N}} = \frac{\sum_{i=2,3,4,10} TTF_i}{\sum_{i=2,3,4,10}(TTF_i + TTR_i)} = 0.616972.$$

As can be seen, software life-extension greatly improves the availability of the VM because of the difference in TTFs. Since the tests are carried out under the accelerated workload by the load client, the observed TTFs are considerably shorter than the TTFs in real life. The potential availabilities are much higher than the computed A_{L} and A_{N} according to the TTFs under the real workload.

4.3.1.3.4. Availability model

From the experimental results, we present a SMP model that describes the general behavior of software life-extension and estimates the availability of VM. As studied in Section 4.3.1.2.3, out-of-memory failure occurs only after starting memory swapping. The state, when a VM is using swap space, can be considered as a failure probable state in a software aging model. Therefore, we present a four states SMP model which represents the behavior of software aging and life-extension as shown in Figure 4.35.

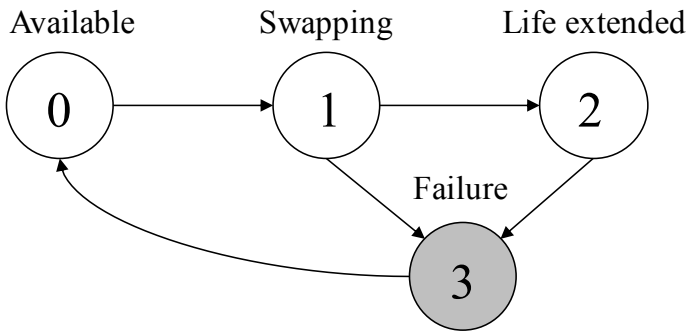


Figure 4.35 Four state SMP model representing the behavior of software life-extension [48]

The model is analogous to the availability model presented in Section 4.3.1.1.4. Unlike the simple CTMC model shown in Figure 4.30(c), the SMP does not limit the distributions of

state transition times to be exponential distribution and hence we can model the system behavior more precisely.

The VM starts in an available state which is represented by State 0. When the VM starts using swap space, the state changes to State 1. If software life-extension is performed successfully before the VM failure, the state changes to State 2. Regardless of whether software life-extension is performed or not, the VM eventually fails and enters in State 3. After manual recovery operation, the state is returned to State 0. Although the VM instance after the recovery operation is not identical to the failed VM, we consider them in a single SMP model under the assumption that each VM instance follows the same state transition.

The SMP can be specified by a state transition probability matrix $P = \{p_{j,k}\}, 0 \leq j, k \leq 3$ and sojourn time distributions $H_j(t), 0 \leq j \leq 3$. Let c be the coverage of software life-extension (i.e., the success probability of life-extension), the transition probability matrix is given by

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & c & 1-c \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Consider the steady-state probability vector $v = [v_0, v_1, v_2, v_3]$ which satisfies the equations $v = vP$ and $\sum_j v_j = 1$. Solving the system of equations, we get

$$v = \left[\frac{1}{3+c}, \quad \frac{1}{3+c}, \quad \frac{c}{3+c}, \quad \frac{1}{3+c}\right].$$

Using the two-stage method [70], the steady-state probability vector $\pi = [\pi_0, \pi_1, \pi_2, \pi_3]$ of the SMP is computed by

$$\pi_j = \frac{v_j h_j}{\sum_{k=0}^{3} v_k h_k}, \qquad j \in \{0,1,2,3\},$$

where h_j represents the mean sojourn time in State j defined:

$$h_j = \int_0^{\infty} \left(1 - H_j(t)\right) dt, \qquad j \in \{0,1,2,3\}.$$

Now we fit the measurement results shown in Table 4.8 to the SMP model to estimate the steady-state availability. First, the coverage of software life-extension is determined to be 0.6 which is the detection coverage. Regarding the sojourn time distributions, TTS and TTR correspond to the sojourn times in State 0 and State 3, respectively. We compute the sojourn times for State 1 by

$$\begin{cases} TTF_i - TTS_i & i = 2,3,4,10 \\ TTL_i - TTS_i & i = 1,5,6,7,8,9, \end{cases}$$

where TTS_i and TTL_i represent the TTS and TTL observed in the life test i, respectively. The sojourn time in State 2 is computed by $TTF_i - TTL_i, i = 1,5,6,7,8,9$. Although we do not know the exact distributions of sojourn times, the mean sojourn time h_j can be estimated by sample mean. The estimated mean sojourn times are computed

$$\hat{h} = \left[\hat{h}_0, \hat{h}_1, \hat{h}_2, \hat{h}_3\right] = [74.2, \quad 262.6, \quad 1176.5, \quad 197.0].$$

Since the VM is available in the all states except State 3, the steady-state availability is given by

$$\hat{A}_{\mathrm{L}} = \sum_{j=0}^{2} \pi_i = \frac{\sum_{j=0}^{2} v_j h_j}{\sum_{k=0}^{3} v_k h_k} = \frac{h_0 + h_1 + c \cdot h_2}{h_0 + h_1 + c \cdot h_2 + h_3}$$

Substituting the variables with the estimated coverage and the mean sojourn times $\hat{h}$, we get the estimated steady-state availability $\hat{A}_{\mathrm{L}} = 0.84064$. The result fits well with the empirical availability A_{L} as seen in the previous section *C*.

4.3.1.3.5. User-perceived metrics

Next, we analyze the user-perceived availability and performance. Table 4.9 shows the statistics obtained from the measurement client during the ten times of life tests on memcached.

The request arrival rate can be computed by the total number of accesses divided by the total test time T=12398 [sec]. However, the computed request arrival rate 0.9545 disagrees with the expected request rate λ_{mes}=1.0 generated by the measurement client. This is caused

by the implementation of the measurement client which employs *closed system model* [89] for request generation. In a closed system model, new request is generated only after the completion of the precedent request. During the period of the VM failure, our measurement client waits for the request time-out which results in a delay of the subsequent request. To get a better analysis, we need to separate the arrival rate during the VM *up* states from that in the VM *down* state. Considering the separation, we can compute the effective request arrival rates as shown in Table 4.10. The request arrival rate in the VM up states λ_{up} is close to the original request rate λ_{mes}=1.0.

Table 4.9 Number of requests from measurement clients [48]

Test No.	Num. of requests	Num. of request processed	Num. of request drops
1	1729	1554	175
2	467	333	134
3	515	330	185
4	477	320	157
5	1599	1429	170
6	1675	1507	168
7	1599	1449	150
8	1549	1420	129
9	1698	1566	132
10	526	373	153
Total	11834	10281	1553

Table 4.10 Effective request arrival rates in VM up states and VM down state [48]

Symbol	Value	Description
λ_{up}	0.992579	Request arrival rate in VM up states
λ_{down}	0.789503	Request arrival rate in VM down state

The request arrival rate in the VM down state λ_{down} can be used in the estimation of the number of request drops that is one of user-perceived availability measures. Since the requests arrive at rate λ_{down} when the VM is down, the expected number of request drops during the total test time T is estimated by

$$\widehat{N}_{\text{drop}} = \lambda_{down} \cdot T \cdot \left(1 - \hat{A}_{\text{L}}\right).$$

Using the result of estimated steady-state availability with T=12398, we obtain $\widehat{N}_{\text{drop}} = 1555.32$, which gives a good estimate of the total number of request drops in Table 4.9.

The user-perceived performance can be characterized by cache hit rate of memcached. The observed number of cache hits and cache hit rates are shown in Table 4.11.

Table 4.11 Observed cache hit rates during the life time tests [48]

Test No.	Life extension	Num. of cache hits	Cache hit rate
1	Applied	1469	0.945302
2	Not applied	264	0.792793
3	Not applied	266	0.806061
4	Not applied	260	0.8125
5	Applied	1319	0.923023
6	Applied	1393	0.924353
7	Applied	1336	0.922015
8	Applied	1304	0.91831
9	Applied	1448	0.924649
10	Not applied	311	0.83378

As shown in the difference of cache hit rates, the performance is improved by software life-extension (i.e., higher hit rates in the tests 1 and 5-9). In contrast, if we apply software rejuvenation to memcached, all of memory content is cleared at the rejuvenation and it causes considerable cache misses after restart. Software rejuvenation is not preferable in terms of user-perceived performance in this scenario. Figure 4.36 presents the changes of cache hit

rate in the life test 1. The cache hit rate increases over 90% when the lifetime is longer than 650 seconds. If software life-extension is not applied, the lifetime ends after approximately 330 seconds (as observed in TTFs in Table 4.8) and the cache hit rate does not reach 90%.

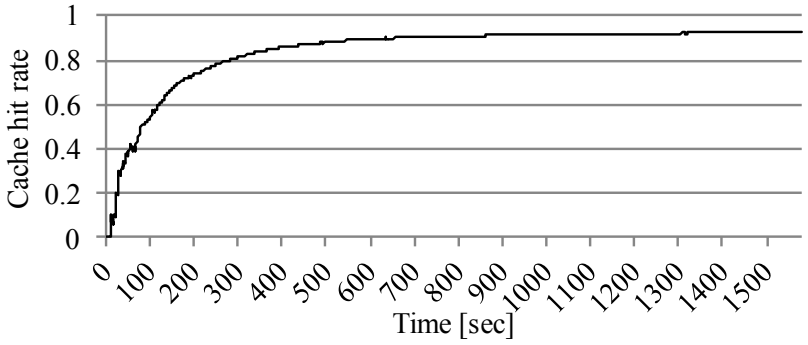


Figure 4.36 Cache hit rate during life test 1 [48]

4.3.1.4. Discussion

While the experiments show the feasibility and effectiveness of software life-extension by dynamic memory allocation to a VM, the approach of software life-extension is not limited to this experimental scenario. As discussed in Section 4.3.1.1.2, workload control is an alternative approach for extending the lifetime of software suffering from aging. As we examine the relationship between the amount of memory allocation and the lifetime in Section 4.3.1.2, for workload control approach, the relationship between workload and lifetime needs to be analyzed. Characterization of workload-aging relationship [85], accelerated degradation tests [39] or accelerated life tests [38] can be used for this purpose.

The VM-based software life-extension appears to be generally applicable to any kind of applications running on VM. However, it limits the applications that can recognize dynamically added memory and make use of them. For example, applications running on Java Virtual Machine (JVM) cannot take the advantage of dynamically added memory because JVM max heap size is set during initialization and it cannot be modified during the execution.

The problem of configuration error on memcached pointed out in our experiments may be removed by allocating sufficiently large amount of memory to VM or by reconsidering the

configuration of memcached. However, the trouble shooting for such configuration errors is not simple as with debugging in software development. We need to perform experiments to reproduce the problem and figure out how large amount of memory is necessary for robust operation. In addition, unlike memcached, applications may not provide configuration parameters for limiting the memory usage. Although software life-extension does not remove the source of a problem, it is useful for mitigating the software aging temporarily in system operation.

4.3.1.5. Related work

The studies of software aging and rejuvenation have been broadly classified into two categories; model-based and measurement-based. The recent survey pointed out that many works are devoted to analytic modeling for rejuvenation scheduling and advocated the necessity of the hybrid (i.e., model and measurement based) approaches [37]. The argument should be true for software life-extension as well. Our experimental study is based on measurements and the observed results are analyzed with analytical model that is an extension of outcomes from model-based studies [44][52][53][54]. While we do not provide the analysis of an optimal trigger for software life-extension in this paper, our preliminary study might open further research direction of the optimal scheduling analysis (e.g., combination of software rejuvenation and life-extension).

Since software life-extension allows software to continue its execution even under software aging states, the concept has a similarity to failure-oblivious computing [90] which continues the software execution even after system error by neglecting the error. Google admits the effectiveness of failure-oblivious computing in their parallel data analysis with Sawzal [91]. Similarly, Rx is presented as a safe survival technique for software failures using checkpoint and re-execution in a modified environment [92]. In contrast to these failure survival techniques, in this paper we focus on software life-extension as a preventive maintenance operation to postpone the time to failure occurrence.

Clustering is the most commonly applied solution of tolerating failures of cache server. Web cache servers are often configured as clusters for high-performance, scalability and

availability [93]. Dynamo [94] and PNUTS [95] keep some essential data in memory and create replicas to be distributed to multiple locations. Recently RAMCloud has presented a quick recovery technique for DRAM-based storage system by scattering backup data across several disks and harnessing hundreds of servers to reconstruct the lost data [96]. While clustering or replication requires a careful design and deployment of system with network configuration, our solution provides a simple countermeasure to a failure of cache server that can be performed manually in the operation.

4.3.1.6. Summary

In this section, we presented a new countermeasure to software aging which extends the lifetime of software execution before encountering a failure caused by resource exhaustion. The feasibility of software life-extension is studied in the scenario of dynamic resource allocation to a virtual machine hosting a memcached which suffers from memory aging due to wrong configuration of maximum memory consumption. We have observed that the lifetime of memcached was greatly improved even by a small amount of additional memory allocation. Compared with software rejuvenation, software life-extension is especially preferable for the applications storing essential data in memory storage because it keeps memory content as long as possible during the lifetime. Through the experiments on memcached, we studied the effectiveness of software life-extension in terms of system availability, the number of request drops and cache hit rate.

4.3.2. Optimal schedule for software life-extension

In this section, we further investigate the effectiveness of software life-extension against software aging [83]. In contrast to the simple Continuous Time Markov Chain model presented in Section 4.3.1.1, we reformulate the system's behavior with a semi-Markov process (SMP). The SMP allows us to use a general distribution for the failure time distribution and to correctly represent the behavior of a time-based software life-extension whose interval is deterministic. By analyzing the SMP, we find the optimum life-extension interval that can maximize the system availability. On the basis of the theoretical foundation

of SMP, we can theoretically clarify the conditions under which the unique optimum life-extension interval exists that have never been addressed in the previous literature. Moreover, we propose an effective hybrid approach in which software life-extension is followed by rejuvenation. The extended SMP that captures the behavior of the system with the hybrid approach is then used to find the optimum combination of intervals for life-extension and rejuvenation that maximizes the system availability. We also show the impact of the software life-extension on the completion time distribution of jobs running on the software system. Through a numerical study, we show that the hybrid approach is a better option than relying on solely either rejuvenation or life-extension in terms of both system availability and job completion time.

4.3.2.1. System model

In order to study the advantages and drawbacks of software life-extension in relation to software rejuvenation, we presented continuous time Markov chain (CTMC) models in the previous paper [48]. The basic assumption of CTMC is that all the state transitions times are exponentially distributed. In this paper, we relax this assumption and devise a more general model using a semi-Markov process (SMP). The state transitions in SMP can follow any type of distribution. This property allows us to represent deterministic trigger for starting preventive operations (software rejuvenation and software life-extension).

In the following subsections, we review the general SMP model for time-based software rejuvenation and the way to get the optimal software rejuvenation interval in terms of system availability. Next, we propose a SMP model for time-based software life-extension in which software life-extension is applied in a deterministic time interval after the latest restart. Interestingly, under specific conditions, software life-extension also has an optimal trigger interval in terms of system availability. We theoretically clarify this point in Section 4.3.2.1.2. Finally, Section 4.3.2.1.3 describes an SMP model for a hybrid approach in which software life-extension is followed by software rejuvenation.

4.3.2.1.1. Time-based rejuvenation model

In 1995, Garg et al. [52] were the first to model time-based software rejuvenation with Markov Regenerative Stochastic Petri Net (MRSPN). Later Chen et al. [56] introduced a three state SMP model that is equivalent to the original MRSPN model.

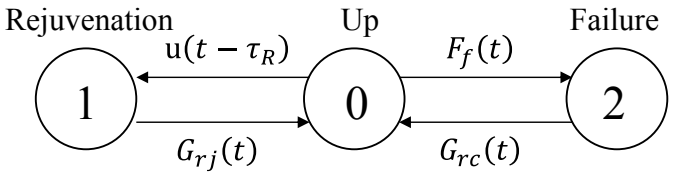


Figure 4.37 SMP representing the system behavior with rejuvenation [83]

Figure 4.37 shows the general three-state SMP model for time-based software rejuvenation (the model is introduced in Section 4.2.1.2 as well). In the previous decade, many researchers used this model to analyze the optimum software rejuvenation trigger interval for maximizing system availability or minimizing the downtime cost (e.g., [56][60]). In Figure 4.37, state 0 is the up state (the software is running). From state 0, the system enters either state 1, i.e., the rejuvenation state, or state 2, i.e., the failure state. The failure time distribution is represented by $F_f(t)$, while the deterministic transition from state 0 to state 1 can be represented by a unit step function $u(t-\tau_R)$, where τ_R is the rejuvenation trigger interval. The recovery time distributions from state 1 and state 2 are represented by $G_{rc}(t)$ and $G_{rj}(t)$, respectively. The steady-state availability of the system A_R is computed as the steady-state probability of state 0 (π_0) [56]:

$$A_R = \pi_0 = \frac{h_0}{h_0 + h_1\left(1 - F_f(\tau_R)\right) + h_2 F_f(\tau_R)}$$

where h_0, h_1 and h_2 are the mean sojourn times for the corresponding states:

$$h_0 = \int_0^{\tau_R}\left(1 - F_f(t)\right)dt\,, h_1 = \int_0^{\infty}\left(1 - G_{rj}(t)\right)dt\,, \qquad h_2 = \int_0^{\infty}\left(1 - G_{rc}(t)\right)dt$$

Dohi et al. [54] showed that expression (1) is strictly convex with respect to τ_R if $F_f(t)$ has the property of increasing failure rate (IFR). Since it is natural to assume that the IFR is

caused by software aging, the above condition is likely to hold. If we assume $\frac{dA(0)}{d\tau_R} > 0$ and $\frac{dA(\infty)}{d\tau_R} < 0$, the steady-state availability is maximized at τ_R^*, which satisfies the following equation.

$$\left(1 - F_f(\tau_R^*)\right)\left[h_1\left(1 - F_f(\tau_R^*)\right) + h_2 F_f(\tau_R^*)\right] - \frac{dF_f(\tau_R^*)}{d\tau_R} h_0(h_2 - h_1) = 0.$$

The maximum steady-state availability is

$$A_R^* = \frac{1 - F_f(\tau_R^*)}{1 - F_f(\tau_R^*) + \frac{dF_f(\tau_R^*)}{d\tau_R}(h_2 - h_1)}.$$

The optimum rejuvenation interval τ_R^* is not represented symbolically. However, it can be obtained by taking a numerical approach as in [60].

4.3.2.1.2. Time-based life-extension model

We construct an SMP model for software life-extension in an analogous way to software rejuvenation. The system is assumed to be failed with failure distribution $F_f(t)$, which is IFR due to software aging. If we apply software life-extension before a system failure, the system enters a new state whose failure rate must be smaller than the original state. To represent this state transition, we add a new life-extended state to the SMP. The proposed general SMP model is shown in Figure 4.38.

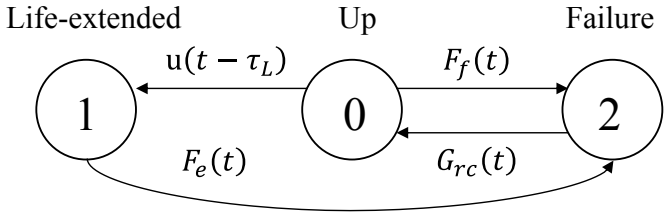


Figure 4.38 SMP representing the system behavior with life-extension [83]

Similar to the rejuvenation model, state 0 and state 2 represent the up and failed states, respectively. State 1 is a life-extended one that has an incoming transition from state 0 and an outgoing transition to state 2. Unlike what happens with software rejuvenation, the system

eventually fails regardless of whether life-extension is applied or not. In other words, software execution ends only in state 2. However, we can extend the lifetime of the software execution at an appropriate time. The failure time distribution changes from state 0 to state 1. We denote the failure time distribution in state 1 as $F_e(t)$. If the software life-extension is applied in a deterministic time interval τ_L, the distribution from state 0 to state 1 can be represented as a unit step function $\mathrm{u}(t-\tau_L)$. We use the same distribution $G_{rc}(t)$ for the recovery transition from state 2. Our previous study [48] assumed that the failure time distribution is hypo-exponential, but the presented SMP model allows $F_f(t)$ to be a general distribution. The SMP model is a generalization of our previous model for software life-extension. Note that the Markov property in state transition from state 0 to state 1 may impact on the applicability of the model if the failure time distribution $F_e(t)$ depends on the time spent in the state 0. We will discuss such a special case in the later Section.

Using the two-stage method for SMP [70], the steady-state availability of the system can be computed from the sum of steady-state probabilities for state 0 and state 1:

$$A_L = \pi_0 + \pi_1 = \frac{h_0 + h_1\left(1 - F_f(\tau_L)\right)}{h_0 + h_1\left(1 - F_f(\tau_L)\right) + h_2}$$

where

$$h_0 = \int_0^{\tau_L}\left(1 - F_f(t)\right)dt\,, \qquad h_1 = \int_0^{\infty}\left(1 - F_e(t)\right)dt\,, \qquad h_2 = \int_0^{\infty}\left(1 - G_{rc}(t)\right)dt.$$

Steady-state availability A_L can be considered as a function of τ_L. Define the failure rate function as

$$r_f(t) = \frac{1}{1 - F_f(t)}\frac{dF_f(t)}{dt}.$$

Similar to the analysis of optimum rejuvenation interval reviewed in the previous section, it is interesting to clarify the condition where the value of A_L is maximized. Since the life-extension changes the state to prolong the time to failure, the effectiveness of life-extension relies on the relation between the failure time distribution $F_f(t)$ and $F_e(t)$. The following

theorem indicates that the optimal trigger of life-extension determines by the relation of the failure rate function $r_f(t)$ and the mean sojourn time h_1, which are characterized by $F_f(t)$ and $F_e(t)$, respectively.

Theorem 4.3.2.1. *When the failure time distribution $F_f(t)$ is IFR and the mean sojourn time in the life-extended state h_1 satisfies the inequality, $r_f(0) < 1/h_1 < r_f(\infty)$, there is a unique value τ_L^* that maximizes the value of A_L.*

Proof. In the following proof, we show that $A_L(\tau_L)$ is concave in the range of $\tau_L > 0$ under the given condition. First taking the derivative of $A_L(\tau_L)$ in terms of τ_L, we get

$$\frac{dA_L(\tau_L)}{d\tau_L} = \frac{h_2\left(1 - F_f(\tau_L) - h_1\frac{dF_f(\tau_L)}{d\tau_L}\right)}{\left(h_0 + h_1\left(1 - F_f(\tau_L)\right) + h_2\right)^2}.$$

The sign of the derivative depends on the numerator and especially on the following term,

$$1 - \frac{1}{1 - F_f(\tau_L)} h_1 \frac{dF_f(\tau_L)}{d\tau_L} = 1 - h_1 r_f(\tau_L).$$

The failure rate function $r_f(t)$ is a strictly monotonically increasing function, as the corresponding distribution $F_f(t)$ is IFR. The sign of (9) changes from negative to positive at a certain value in the range of $\tau_L > 0$ under the given condition $r_f(0) < 1/h_1 < r_f(\infty)$. As a result, $A_L(\tau_L)$ is a concave function in $\tau_L > 0$ and the value is maximized at τ_L^* that satisfies $h_1 = 1/r_f(\tau_L)$. □

The optimum interval τ_L^* is not represented symbolically, but the value can be numerically obtained in a similar way as the optimum software rejuvenation interval.

Intuitively, the failure rate in state 0 increases over time, and whenever it reaches the mean failure rate in state 1 $(1/h_1)$, it is the best timing at which to move to state 1. Instead of a decreased failure rate in state 1, there may be a performance penalty after life-extension; this is addressed in the job completion time analysis presented in Section 4.3.2.2.

4.3.2.1.3. Hybrid approach model

Software rejuvenation and software life-extension are not exclusive. Rather, they can be combined together in an epoch of the execution lifecycle. We devised such a hybrid approach in which software life-extension is followed by software rejuvenation. The corresponding SMP is drawn in Figure 4.39.

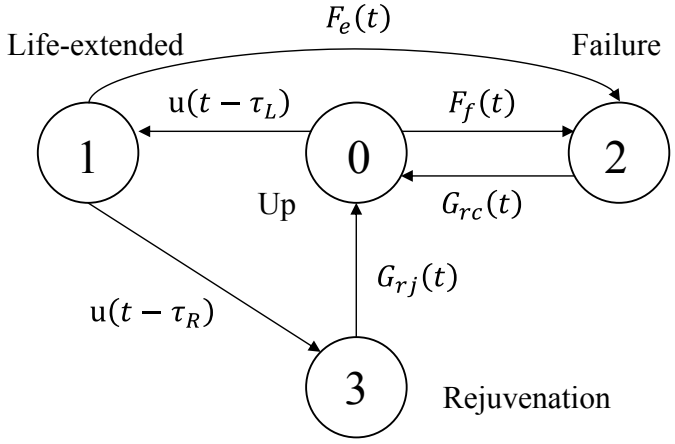


Figure 4.39 SMP representing the system behavior with both rejuvenation and life-extension [83]

The model has both a life-extended state (state 1) and a rejuvenation state (state 3). Software life-extension is applied at time τ_L after the system starts in state 0, while software rejuvenation is applied at time τ_R after the system enters in state 1. The recovery time distributions from state 2 and state 3 are represented by $G_{rc}(t)$ and $G_{rj}(t)$, respectively, and the failure time distribution in state 1 is denoted as $F_e(t)$. In this system, software rejuvenation can be used after a software life-extension so as to minimize the potential downtime. Note that the SMP model asymptotically becomes the time-based life-extension model by taking τ_R to infinity, and it becomes the time-based rejuvenation model by taking τ_R to be 0. In the following discussion, we exclude these extreme cases and assume that $0 < \tau_R < \infty$.

The steady-state availability of the system is the sum of the steady-state probabilities of state 0 and state 1, as computed by

$$A_{LR} = \pi_0 + \pi_1 = \frac{U_{LR}}{U_{LR} + D_{LR}},$$

where

$$U_{LR} = h_0 + h_1\left(1 - F_f(\tau_L)\right),$$

$$D_{LR} = h_2\left[F_f(\tau_L) + \left(1 - F_f(\tau_L)\right)F_e(\tau_R)\right] + h_3\left(1 - F_f(\tau_L)\right)\left(1 - F_e(\tau_R)\right),$$

and

$$h_0 = \int_0^{\tau_L}\left(1 - F_f(t)\right)dt\,, h_1 = \int_0^{\tau_R}\left(1 - F_e(t)\right)dt,$$

$$h_2 = \int_0^{\infty}\left(1 - G_{rc}(t)\right)dt\,, h_3 = \int_0^{\infty}\left(1 - G_{rj}(t)\right)dt$$

We can control both the life-extension interval τ_L and software rejuvenation interval τ_R, and thus, the steady-state availability A_{RL} can be considered to be a bivariate function of τ_L and τ_R. Let us define the hazard rate functions $r_f(t)$ and $r_e(t)$ as

$$r_f(t) = \frac{1}{1 - F_f(t)}\frac{dF_f(t)}{dt}, \quad r_e(t) = \frac{1}{1 - F_e(t)}\frac{dF_e(t)}{dt}.$$

The following theorem is derived for analyzing the optimum combination of life-extension interval and rejuvenation interval that maximizes A_{LR}.

Theorem 4.3.2.2. *When both the failure time distribution $F_f(t)$ and the failure time distribution in life-extended state $F_e(t)$ are IFR, and the nonlinear function $Z = h_1h_2 - \left(1 - F_e(\tau_R)\right)(h_2 - h_3)h_0$ is always smaller than 0, and $h_2 > h_3$, the optimum combination of the life-extension interval τ_L and rejuvenation interval τ_R that maximizes A_{RL} is given by the solution of the following system of equations.*

$$\begin{cases} D_{LR} - r_e(\tau_R)(h_2 - h_3)U_{LR} = 0 \\ D_{LR} + r_f(\tau_L)Z = 0 \end{cases}.$$

Proof. First, we prove that A_{RL} is a strictly concave function in terms of τ_R. Then we show the function is also strictly concave on τ_L under the given condition. The derivative of A_{RL} with respect to τ_R is

$$\frac{\partial A_{LR}}{\partial \tau_R} = \frac{1}{(U_{LR} + D_{LR})^2}\left[\frac{\partial U_{LR}}{\partial \tau_R} D_{LR} - \frac{\partial D_{LR}}{\partial \tau_R} U_{LR}\right]. \tag{19}$$

The derivatives of U_{RL} and D_{RL} with respect to τ_R are

$$\begin{aligned} \frac{\partial U_{LR}}{\partial \tau_R} &= \Big(1 - F_f(\tau_L)\Big)\big(1 - F_e(\tau_R)\big), \\ \frac{\partial D_{LR}}{\partial \tau_R} &= \frac{\partial F_e(\tau_R)}{\partial \tau_R}(h_2 - h_3)\Big(1 - F_f(\tau_L)\Big). \end{aligned} \tag{20}$$

Applying (20) to (19), the sign of (19) is determined by the following term in the numerator

$$W = D_{LR} - r_e(\tau_R)(h_2 - h_3)U_{LR}.$$

Taking the derivative of W with respect to τ_R and using (20), we have

$$\begin{aligned} \frac{\partial W}{\partial \tau_R} &= \frac{\partial D_{LR}}{\partial \tau_R} - (h_2 - h_3)\left[\frac{\partial r_e(\tau_R)}{\partial \tau_R} U_{LR} + r_e(\tau_R)\frac{\partial U_{LR}}{\partial \tau_R}\right] \\ &= -\frac{\partial r_e(\tau_R)}{\partial \tau_R}(h_2 - h_3)\Big(h_0 + \Big(1 - F_f(\tau_L)\Big)h_1\Big) < 0 \end{aligned}$$

Since W is strictly decreasing, A_{RL} is strictly concave in τ_R. The optimum rejuvenation interval is given by τ_R^*, which satisfies W=0. The maximum availability is written as

$$A_{LR}^* = \frac{1}{1 + r_e(\tau_R^*)(h_2 - h_3)}.$$

With the optimal rejuvenation interval τ_R^*, we consider A_{LR} as a function of τ_L and take the derivative with respect to τ_L:

$$\frac{\partial A_{LR}}{\partial \tau_L} = \frac{1}{(U_{LR} + D_{LR})^2}\left[\frac{\partial U_{LR}}{\partial \tau_L} D_{LR} - \frac{\partial D_{LR}}{\partial \tau_L} U_{LR}\right].$$

From the numerator, the sign of the derivative is determined by

$$V = D_{LR} + r_f(\tau_L)Z$$

Taking the derivative of V with respect to τ_L, we have

$$\frac{\partial V}{\partial \tau_L} = \frac{\partial D_{LR}}{\partial \tau_L} + \frac{\partial r_f(\tau_L)}{\partial \tau_L} Z - r_f(\tau_L) \frac{\partial Z}{\partial \tau_L} = \frac{\partial r_f(\tau_L)}{\partial \tau_L} Z < 0.$$

Under the given condition, V is strictly decreasing in τ_L and A_{LR} is strictly concave in τ_L as well. The availability A_{LR} is maximized when V=0. Consequently, the optimum combination of τ_L and τ_R which maximizes A_{LR} is given by the solution of the system of equations W=0 and V=0. □

The optimum life-extension interval and rejuvenation interval cannot be expressed in a symbolic manner, but they can be computed through a numerical approach like a gradient search method used for analyzing the optimum rejuvenation intervals in a virtualized system [12]. Numerical examples are presented in Section 4.3.2.3.

4.3.2.2. Job completion time analysis

In this section, we analyze the distribution of completion times of jobs running on the software system based on the SMPs presented in the previous section. As a result of the preventive maintenance operations like software rejuvenation and life-extension, the software execution status changes and the executing job is preempted at the beginning of the new state. Performing software rejuvenation causes a preemptive-repeat (PRT) [64] discipline in which the job restarts from the beginning. In contrast, software life-extension does not lose its execution status; it thus follows a preemptive-resume (PRS) [64] discipline wherein the job resumes at the point it was preempted. The job completion time to process all the requested work is clearly affected by the preemption type as well as the state transitions. Here, we will perform a job completion time analysis on aging software without any preventive operations and one on software with a time-based life-extension.

4.3.2.2.1. Aging software

If the software system suffering from aging is not controlled with any preventive operations, its behavior can be captured by a two state model composed of an up state and a down state.

Let $F_f(t)$ and $G_{rc}(t)$ denote the failure time distribution and recovery time distribution. Since the job execution is dropped when the system goes down, the down state is considered to be a PRT state. Once the job is interrupted in the down state, it needs to be restarted after recovery.

Define $\mathrm{T}(x)$ to be the amount of time needed to complete a job with a work requirement of *x* units. Suppose that the execution environment processes a work unit in an hour, $\mathrm{T}(x)$ is equal to *x* if the job started in the up state completes before the first failure occurrence. If the job encounters a failure at time *h* (>0), the job needs to be restarted after the system recovers. In this case, the total job execution time becomes the sum of *h*, recovery time, and $\mathrm{T}(x)$. Taking the Laplace-Stieltjes transform (LST) with respect to *t*, the LST of the job completion time $\Phi_T^{\sim}(s,x)$ satisfies the following equation:

$$\Phi_T^{\sim}(s,x)|_h = \begin{cases} e^{-sx}, & h \geq x \\ e^{-sh} G_{rc}^{\sim}(s) \Phi_T^{\sim}(s,x), & h < x \end{cases}$$

where $G_{rc}^{\sim}(s)$ is the LST of $G_{rc}(t)$. Unconditioning on *h*, $\Phi_T^{\sim}(s,x)$ is expressed as

$$\begin{aligned} \Phi_T^{\sim}(s,x) &= e^{-sx} \int_x^{\infty} dF_f(h) + G_{rc}^{\sim}(s) \Phi_T^{\sim}(s,x) \int_0^x e^{-sh} dF_f(h) \\ &= \frac{e^{-sx}\left(1 - F_f(x)\right)}{1 - G_{rc}^{\sim}(s) \int_0^x e^{-sh} dF_f(h)}. \end{aligned} \tag{21}$$

The expected job completion time is computed from the moment generation property of LST [70]:

$$E\big(T(x)\big) = -\frac{\partial \Phi_T^{\sim}(s,x)}{\partial s}\bigg|_{s=0}. \tag{22}$$

Now let us consider a system using software rejuvenation to prevent a failure caused by software aging. The system state transition can be captured by the SMP shown in Section 4.3.2.1.1. We assume that the job execution begins immediately after restarting the execution environment (in state 0). In this system, the rejuvenation time interval τ_R must be larger than the work requirement *x*; otherwise, the job never completes. Under this condition $\tau_R > x$, the job completion time distribution is the same as that without rejuvenation. Therefore,

the LST of the job completion time is represented by (21), and the expected job completion time can be computed from (22).

4.3.2.2.2. Job completion time in the case of life-extension

Next, we analyze the job completion time on a system using life-extension. The system behavior follows the state transitions specified in Section 4.3.2.1.2. Again, we assume that the job execution begins just after restarting the execution environment (in state 0). When the life-extension interval τ_L is larger than x, the job never runs in a life-extended state (State 1), and the job completion time is the same as in the case of the aging system studied in Section 4.3.2.2.1. In the case of $\tau_L \leq x$, there are two scenarios in which the job is dropped as a result of a software failure: the software execution fails 1) before life-extension and 2) after life-extension.

Conditioned by the failure time h, the LST of job completion time is represented by

$$\Phi_{TL}^{\sim}(s,x)|_h = \begin{cases} e^{-sx}, & h \geq x, \\ e^{-sh} G_{rc}^{\sim}(s) \Phi_{TL}^{\sim}(s,x), & h < x. \end{cases}$$

Because the failure time distribution changes to $F_e(t)$ after software life-extension at $t = \tau_L$, the LST of the job completion time $\Phi_{TL}^{\sim}(s,x)$ is

$$\begin{aligned} \Phi_{TL}^{\sim}(s,x) &= e^{-sx}\left(1 - F_f(\tau_L)\right)\int_x^{\infty} dF_e(h-\tau_L) \\ &\quad + G_{rc}^{\sim}(s)\Phi_{TL}^{\sim}(s,x)\left[\int_0^{\tau_L} e^{-sh} dF_f(h) + \left(1 - F_f(\tau_L)\right)\int_{\tau_L}^{x} e^{-sh} dF_e(h-\tau_L)\right] \\ &= \frac{e^{-sx}\left(1 - F_f(\tau_L)\right)\left(1 - F_e(x-\tau_L)\right)}{1 - G_{rc}^{\sim}(s)\Psi_{TL}^{\sim}(s,x)}, \end{aligned} \tag{23}$$

where

$$\Psi_{TL}^{\sim}(s,x) = \int_0^{\tau_L} e^{-sh} dF_f(h) + \left(1 - F_f(\tau_L)\right)\int_{\tau_L}^{x} e^{-sh} dF_e(h-\tau_L)$$

The above analysis does not take into account the performance degradation after the software life-extension. If the job processing rate decreases in the life-extended state (i.e., State 1) by $r(0 < r \leq 1)$, $\Phi_{TL}^{\sim}(s,x)$ is expressed as

$$\Phi_{TL}^{\sim}(s,x) = \frac{e^{-s\left(\tau_L+\frac{x-\tau_L}{r}\right)}\left(1-F_f(\tau_L)\right)\left(1-F_e\left(\frac{x-\tau_L}{r}\right)\right)}{1-G_{rc}^{\sim}(s)\Psi_{TL}^{\sim}(s,x)}$$

where

$$\Psi_{TL}^{\sim}(s,x) = \int_0^{\tau_L} e^{-sh}dF_f(h) + \left(1-F_f(\tau_L)\right)\int_{\tau_L}^{\tau_L+\frac{x-\tau_L}{r}} e^{-s\left(\tau_L+\frac{h-\tau_L}{r}\right)}dF_e(h-\tau_L).$$

The above expression is a generalization of (23) and it becomes identical to (23) when r=1. The expected job completion time is obtained as

$$E\big(T(x)\big) = -\left.\frac{\partial\Phi_{TL}^{\sim}(s,x)}{\partial s}\right|_{s=0}. \tag{24}$$

The job completion time distribution in the hybrid system model in Section 4.3.2.1.3 is also characterized by (24), provided that the sum of τ_L and τ_R/r is larger than or equal to the work requirement x. If $\tau_L+\frac{\tau_R}{r} < x$, the rejuvenation always takes place before job completion and hence the job can never complete.

4.3.2.3. Numerical example

Our numerical experiments aim to compare the software rejuvenation, life-extension, and hybrid approaches. For the software system suffering from software aging, we first analyze the optimum intervals for software rejuvenation and life-extension that maximize the steady-state system availability. Next, we study the impact on the job completion time on the basis of the model presented in the previous section.

Since the failure rate affected by software aging increases over time, we assume that the failure time distribution is IFR. The hypo-exponential distribution is known to be an IFR distribution regardless of its parameter values, and it has been widely used for modeling software aging (e.g., [44][52][53]). We thus define the failure time distributions as two-stage hypo-exponential distributions, $F_f(t) = HYPO(\lambda_1,\lambda_2)$ and $F_e(t) = HYPO(\lambda_1,\lambda_3)$ where λ_1, λ_2 and λ_3 are parameters that satisfy $\lambda_2 > \lambda_3$. The recovery time distribution and rejuvenation time distribution are assumed to be exponential with rates μ and α, respectively.

The parameter values used in the examples are summarized in Table 4.12; most of them are taken from [63].

Table 4.12 Default parameter values [48]

Parameters	Values	Descriptions
λ_1	0.002976190 [1/h]	
λ_2	0.005952381 [1/h]	Parameters for failure time distributions
λ_3	0.001984127 [1/h]	
μ	1 [1/h]	Reactive recovery rate
α	12 [1/h]	Rejuvenation rate
x	360 [units]	Amount of work requirements

4.3.2.3.1. Steady-state availability

Figure 4.40 plots the steady-state availability achieved by time-based software rejuvenation and time-based software life-extension for default parameter values and varying the maintenance intervals. There exists an optimum rejuvenation interval, since $F_f(t)$ is IFR, $\mu < \alpha, \frac{dA(0)}{d\tau_R} > 0$, and $\frac{dA(\infty)}{d\tau_R} < 0$ [54]. From Theorem 4.3.2.1, an optimum life-extension interval also exists. Table 4.13 lists the optimum rejuvenation and life-extension intervals and the corresponding steady-state availabilities.

Table 4.13 Optimum time interval and maximum availability [48]

Operation	Optimum interval [hours]	Maximum availability
Rejuvenation	144.936	0.99858628
Life-extension	99.612	0.99886750

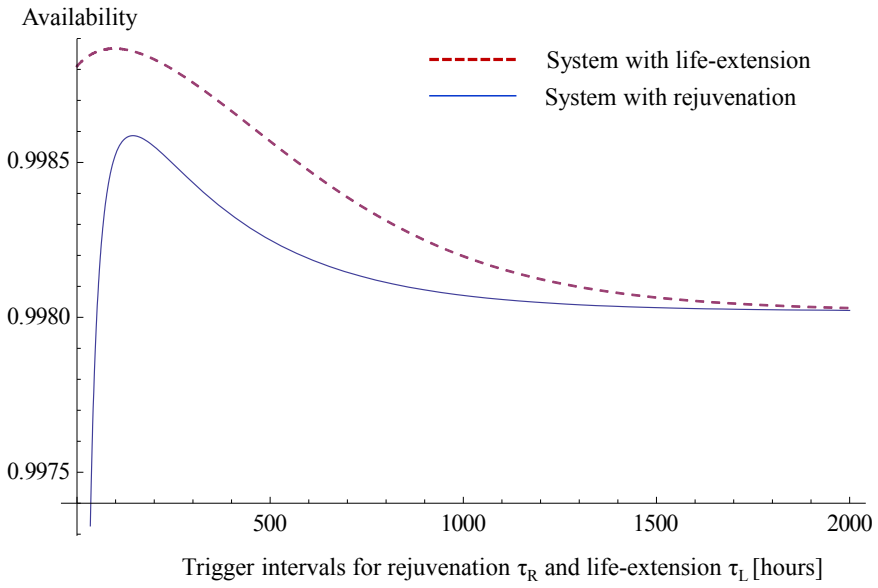


Figure 4.40 Steady-state availabilities achieved by software rejuvenation or life-extension [83]

As can be seen, rejuvenation and life-extension have different optimum intervals that maximize system availability. With the default parameter values, software life-extension achieves higher availability than rejuvenation. However, software rejuvenation potentially achieves higher availability than life-extension, for example when the rejuvenation rate α is high. The impacts of α and λ_3 on the maximum system availability are analyzed by a sensitivity analysis below.

For a system using only software rejuvenation, the optimum rejuvenation interval depends on the rejuvenation rate α. Figure 4.41 plots the maximum availability versus α, where each plot is labeled with the optimum interval τ_R^*.

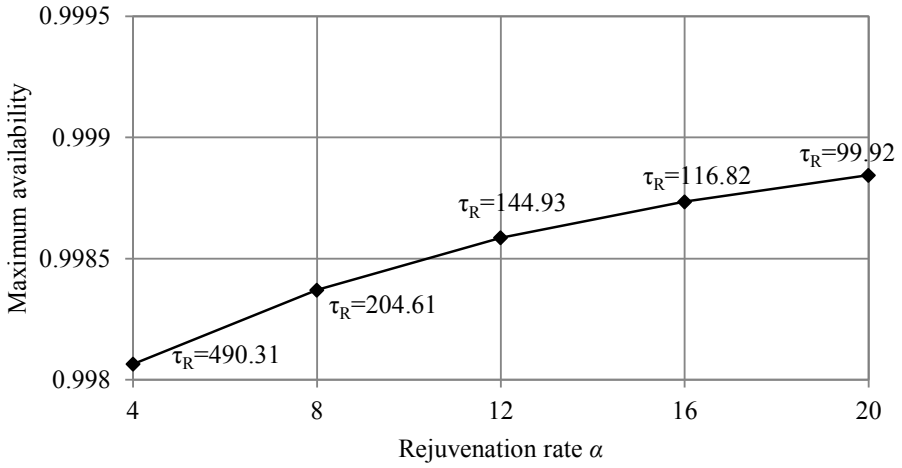


Figure 4.41 Sensitivity to rejuvenation rate on maximum availability [83]

As α increases, the downtime overhead due to rejuvenation decreases; thus, the optimum rejuvenation interval becomes shorter and the maximum steady-state availability increases. Similarly, the optimum life-extension interval depends on λ_3, which determines the failure time distribution in the life-extended state. Since the mean time to failure given $HYPO(\lambda_1, \lambda_3)$ is $\frac{1}{\lambda_1} + \frac{1}{\lambda_3}$, a larger λ_3 shortens the lifetime.

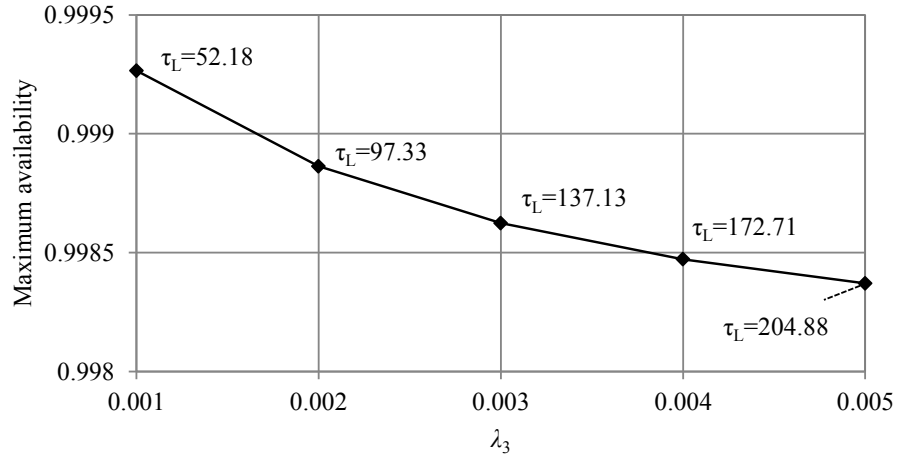


Figure 4.42 Sensitivity of maximum availability to parameter λ_3 [83]

Figure 4.42 plots the maximum availability values achieved by the optimum life-extension intervals τ_L^* versus λ_3. As λ_3 increases, the optimum life-extension interval gradually increases and the maximum availability consequently decreases

Next, we consider the hybrid model that contains two control parameters: a life-extension interval τ_L and rejuvenation interval τ_R. Figure 4.43 plots the steady-state availability values computed by varying τ_L and τ_R from one hour to 2000 hours.

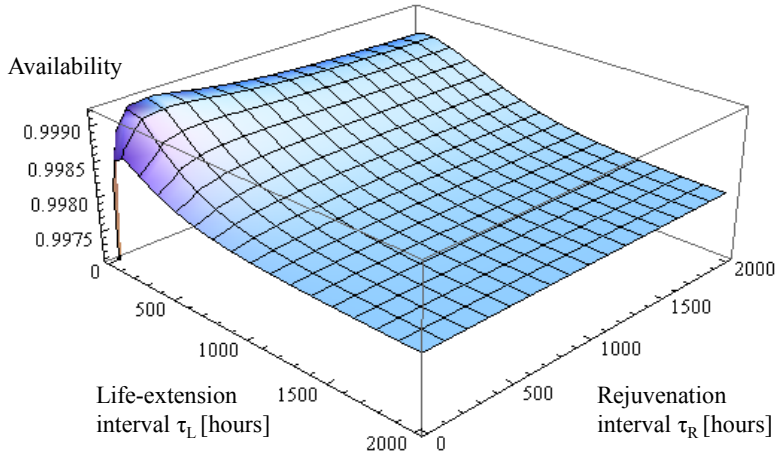


Figure 4.43 Steady-state system availability with hybrid approach [83]

From Theorem 4.3.2.2, there exists a unique optimum combination of τ_R and τ_L at which steady-state availability is maximized. In this example, the maximum steady-state availability is 0.99926 which is achieved at $(\tau_L, \tau_R) = (52.7, 207.9)$. Thus, the hybrid approach potentially has higher system availability compared with the individual approaches.

4.3.2.3.2. Job completion time

Suppose the software system starts processing a job with work requirements x at the beginning of the up state. The job completion time distribution can be characterized by either (21) or (23) depending on the preemption type. Since $\Phi_T^{\sim}(s,x)$ and $\Phi_{TL}^{\sim}(s,x)$ are in LST form, we take numerical inversion using a Mathematica library [97]. We set the performance degradation rate r in the life-extended state to 1.0, 0.8, or 0.5 and compare the results.

Figure 4.44 shows the resulting job completion time distributions for the aging system without life-extension and the one using software life-extension with different degradation rates. The life-extension interval is set to 99.612, which is the optimum interval obtained in Section 4.3.2.3.1.

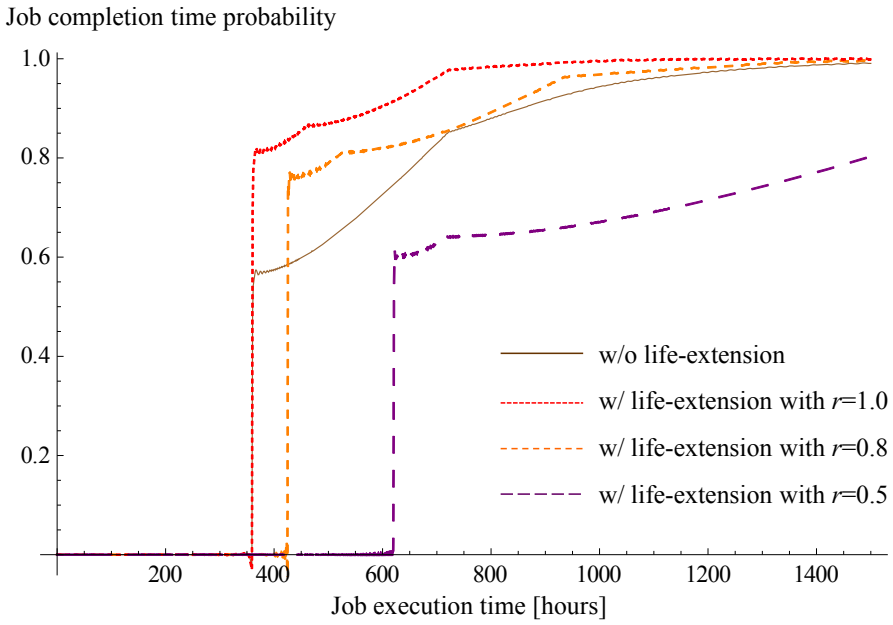


Figure 4.44 Job completion time distributions for aging system and the system with life-extension [83]

If there is no performance degradation after the life-extension (i.e., r=1.0), the system with life-extension can clearly complete the job faster than the system without life-extension. Note that it takes at least 360 hours to complete the job, since the work processing rate is 1 unit/hour in both the cases. When the performance degradation occurs after life-extension, the minimum job completion time is prolonged accordingly. The more significant the degradation in processing rate, the longer the minimum job completion time becomes. Although life-extension has still an advantage when r=0.8, it is no more beneficial than the system without life-extension when r=0.5. This means re-execution after a failure without life-extension is more effective than extending the software execution by reducing the

processing rate. Therefore, the performance degradation rate r is critical to designing an efficient software life-extension as far as the job completion time is concerned.

Since the completion time performance is influenced by the life-extension interval τ_L, the impact of the interval can be evaluated in terms of the mean job completion time (24). Figure 4.45 plots the mean job completion times versus τ_L.

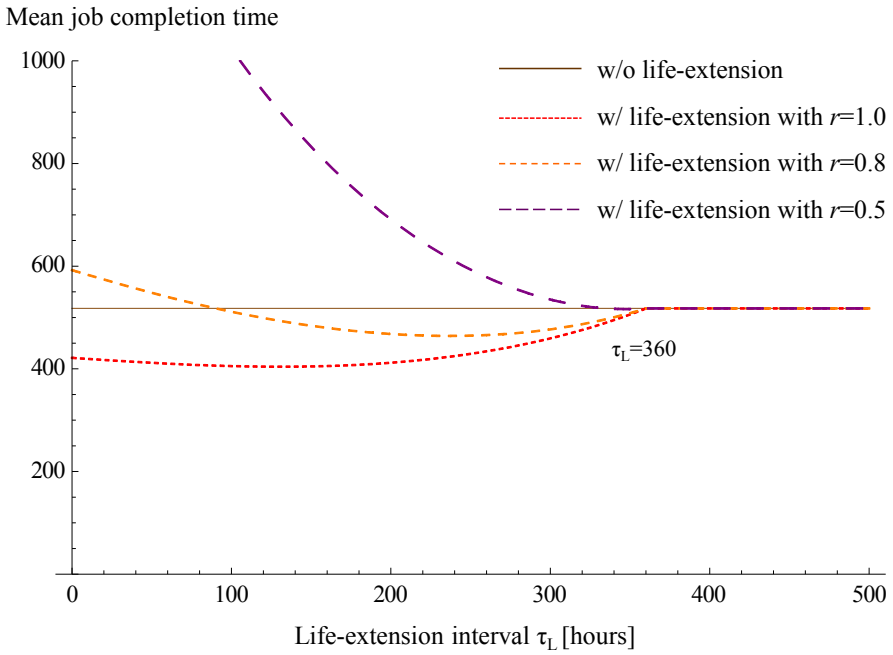


Figure 4.45 Mean job completion time versus life-extension interval [83]

When τ_L is larger than $x = 360$, life-extension is never applied during the job execution, and hence, the mean job completion time is the same as the mean of the aging system without life-extension (517.808 hours). Interestingly, if $\tau_L \leq x$, there is an optimum life-extension interval that minimizes the mean job completion time. If there is no performance degradation (r=1.0), the mean job completion time is minimized at $\tau_L = 130.25$, at which point the job completion time is 404.2 hours. This is the optimum life-extension interval in terms of the mean job completion time. In the case of the performance degradations with r=0.8 and 0.5, the optimum intervals are $\tau_L = 236.49$ and 347.53, and the corresponding minimum mean job completion times are 464.19 and 516.42 hours.

The results of our numerical experiments show that the optimum intervals for preventive maintenance (rejuvenation, life-extension or both) differ depending on the measure of interest (system availability or mean job completion time) as well as the parameter values. The proposed analytical approaches are useful for designing systems with those maintenance operations.

4.3.2.4. Discussion

Our life-extension model introduces a life-extended state that is distinct from the original state and assumes that the failure time distribution changes in the new state. SMP allows us to use any kind of distribution for failure time distribution. However, as mentioned previously, the Markov property assumed on every state transition in SMP may restrict the applications of the proposed model. The amount of accumulated errors in the robust state is not conserved in the life-extended state. If the failure time distribution depends on the time spent in the robust state, one approach we may take is to expand the model by reliability-conservation principle [98]. According to this principle, the reliability at the time of life-extension, $1 - F_f(\tau_L)$, is preserved in the extended state. Let $F_{f2}(t)$ be the failure time distribution in the life-extended state. There exists a time $\widehat{\tau_L}$ that satisfies $1 - F_f(\tau_L) = 1 - F_{f2}(\widehat{\tau_L})$. The failure time distribution when software life-extension is applied at τ_L is represented as $F_{f2}(t + \widehat{\tau_L})$ that depends on the time spent in the robust state τ_L. Note that the resulting state model is no longer SMP, since the process does not regenerate in the life-extension state (i.e., the transition probabilities in the life-extension state depend on the time spent in the previous states). Alternatively, we may also rely on approximation model instead of explicitly modeling the time-dependent failure distribution. The approximated SMP model can be constructed from empirical data for lifetime in the life-extended state. Once we obtain the approximated SMP model, we can apply the same optimization scheme.

Another important future direction is to extend our model by incorporating the cost of software life-extension. In the current model, the cost factor is indirectly included as performance degradation of the life-extended application. The cost of life-extension depends on the implementation (e.g., additional resource usage costs, performance degradations, and

degradations to the availability of other services). Incorporating such factors explicitly in the model can yield more comprehensive optimization problem. For instance, if the server memory is shared by another application on the same server, the life-extension may reduce the memory allocation to other application which results in performance degradation. To incorporate such cost for determining the optimum life-extension interval, we need to model the state transition and performance of the other application in addition to the life-extended application.

4.3.2.5. Summary

We have analytically shown the effectiveness of counteracting software aging by extending the lifetime of software execution. On the basis of the semi-Markov process capturing the behavior of the system, we show of the condition where there is an optimum software life-extension interval in which the system availability is maximized. Our numerical experiments reveal that life-extension is comparable to rejuvenation in terms of system availability, while it can significantly shorten the job completion time. Considering both of system-availability and job completion time performance, the hybrid approach in which software life-extension followed by rejuvenation turns out to be the best strategy.

Chapter 5

Storage array and data backup

The main focus of this chapter is data availability. In the previous chapter, the issue of software aging causing system unavailability and degraded performance was addressed. The effectiveness of software rejuvenation and life-extension techniques has been evaluated with respect to achievable system availability. Data is another critical part of computing in IT systems. With the proliferation of big data analytics and advancement of machine learning technologies, software programs and applications on IT systems are becoming highly data-intensive. From the application perspectives, data availability is crucial for successful execution of a program. Missing or unavailable data access causes a failure of execution or a wrong analysis result. There are many threats of making data unavailable, for example storage system failure, network failure, data corruption, and data lost by operation miss etc. To provide dependable IT systems, it is important to carefully design a data management architecture against several threats to data unavailability. In this chapter, first an overview of data management design is described. There are several techniques and methods of protecting data according to different types of threats. This thesis particularly focuses on the fault-tolerant technique for storage array against disk failure and data back-up operation for data protection. Section 5.2 presents the performability model for RAID storage array that could be useful to find the best configurations of RIAD storage array under given requirements for data availability and performance. Section 5.3 presents an MDP approach to determine the optimum data backup scheduling that must satisfy the given data recovery objectives.

5.1. Data management

As mentioned earlier, it is important to carefully design of a data management architecture to provide dependable IT systems. Usually, systems are required to achieve high data availability and high-performance within a limited budget. The optimum design of data management needs to trade-off these conflicting objectives by selecting relevant technologies and configuration options.

At least four system components must be taken into account in the design of data management; *storage systems*, *middleware*, *application programs* and *data management operation*. A *storage system* is a device to store the digital data in a persistent or non-persistent media. Hard disk drive (HDD) and Solid-state drive (SSD) are used in computers for persistent data storage, while RAM is used as non-persistent storage. The failure of these devices can directly cause data loss; hence, several fault-tolerant techniques, such as RAID array, are used in high-end practical storage systems. *Middleware* is the front-end software for a storage system to provide data accessibility to user applications. Database management systems (DBMSs) and filesystems have been widely used in many computing systems. In the cloud era, many distributed data stores are also used for applications on cloud computing. *Application programs* are the consumers of the data and may also produce data through implemented algorithms or analytic functions. Middleware needs to intermediate both data retrieval and data insertion requests from applications to a storage system. A *data management operation* is carried out using management tools for system administrators so that data are stored in the correct place and accessible for authorized users. Data backup, snapshot, archiving, and restoration from backup data are examples of such management operations. Some operations can be automated using the function implemented in middleware. In many cases, however, data management operations need to be designed across different storage systems with middleware instances that may need further assistance from administration tools.

Due to the diversity of storage systems, middleware, and application workloads, an entire data management system is becoming considerably complex, which results in the boost in demand for streamline data management design. Model-based data management design, as illustrated in Figure 5.1, is a promising solution to this issue.

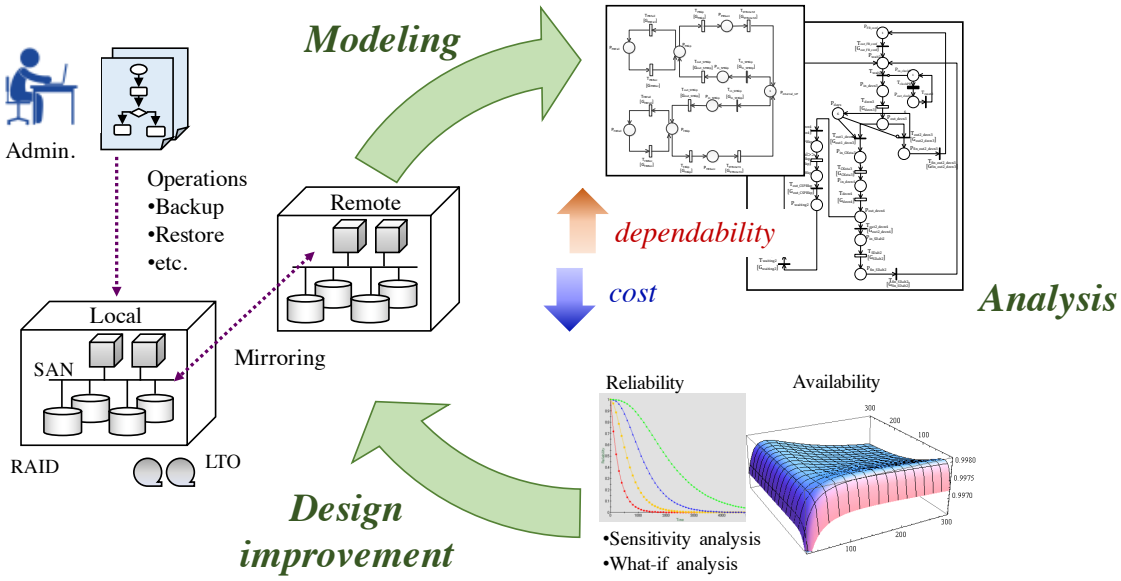


Figure 5.1 Model-based data management design

The target system consists of these inter-connected components. Stochastic models can be applied to capture the configurations and dynamics of data management on the target system, and their effectiveness can be validated through analysis in terms of some quantitative measures (e.g., data availability, performance, cost, etc.). The analytical results can provide a guide to improve the design of storage configuration and data management operations. The impact of some changes can also be evaluated through model-based analysis. System designers can explore the optimum data management architecture through this cycle.

Toward this goal, the following chapters introduce our studies on performability design of storage systems and optimum scheduling for data backup operation.

5.2. Storage system

5.2.1. Performability analysis for RAID storage systems

In this section, we present a performability model for RAID storage systems using Markov regenerative process to compare different RAID architectures [99]. The work has been done in collaboration with Dr. Ruofan Xia and Prof. Kishor Trivedi in Duke University.

While homogeneous Markov models are extensively used for reliability analysis of RAID storage systems, the memory-less property of the sojourn time assumed in such models is not satisfied in reality, especially in disk rebuild process whose progress is not interrupted even at an event of another disk failure. In this study, we use Markov regenerative process which allows us to model the generally distributed rebuild times providing a needed extension of the traditional Markov models. The Markov regenerative process is then used to assess the performability of the storage system by assigning reward rates to each state based on the real storage benchmark results. Our numerical study characterizes the performability advantage of RAID6 architecture over RAID10 architecture in terms of sequential read access. Our findings include that the effect of exponential assumption for the rebuild times has practically negligible effect when we focus on data availability. However, the effect this approximation on performability prediction may not be negligible especially when the performance level drastically changes in degraded states. Our MRGP model provides more accurate prediction of performability in such cases.

5.2.1.1. Introduction

Data availability is considered an essential aspect of recent dependable storage systems. RAID is a well-established popular solution to protect the integrity of data from disk failure events in storage systems. There are several choices of RAID architectures depending on the number of disks constituting the disk arrays as well as the reliability requirement. Assuming exponential distributions for the time to disk failure and disk rebuild time, reliability of RAID storage systems is often evaluated by the measure called mean time to data loss (MTTDL) that can be computed from continuous time Markov chain (CTMC) models. The quantification of reliability of RAID enables designers to select a proper architecture.

In the literature presenting the disk failure statistics, however, it was revealed that the time to disk failure event does not fit exponential distribution. Schroeder et al analyzed 100,000 disk failures and showed that the annual disk failure rate (AFR) is much higher than the expected values derived from MTTF with exponential assumption [100]. Disk failures do not only consist of operational failures such as bad servo-track and bad electronics, but also

include latent defects caused by writing errors or media degradation [101]. The impact of latent sector errors during a disk rebuilding process is not negligible because it may cause a double disk failure leading to data loss. Researchers have made efforts to narrow the gap between the conceptual storage model and the real failure data by improving models and introducing efficient solution methods [101][102][103].

While MTTDL metric and the assumption of exponential distributions are the subjects of criticism [104][105], there are counterarguments that MTTDL is still useful for comparing solution techniques and different architectures of RAID systems. Venkatesan and Iliadis showed that the MTTDL was practically insensitive to the actual failure distribution when failure rate is much smaller than repair rate [106]. Authors in [107] advocate that no study in the literature disproves the validity of MTTDL as a criterion in the comparison of the reliability of one scheme with that of another. Furthermore, the misconceptions in the criticism of using MTTDL are refuted in [108]. We will support this argument and use Markov models for a comparative study of RAID storage systems. However, when we assume the exponential distribution for disk rebuild time, the impact of memoryless property on data availability may not be negligible, as the sensitivity analysis in [106] points out. In order to compare the data availabilities achieved by different RAID architectures, the time to rebuild a disk should be treated precisely in the models.

In this study, we extend the traditional CTMC models for RAID storage systems to deal with non-exponential disk rebuild time. When a disk fails in a RAID system that could tolerate more than one disk failure, a rebuild process starts to reconstruct the data on a spare disk using the data stored in the remaining disks. Even if another disk failure occurs during the rebuild process, the rebuild process continues the operation until the data is reconstructed completely. Since the rebuild times are assumed not to have memoryless property, we might be tempted to use a semi-Markov model. However, this will be incorrect since during the rebuild process, the stochastic model representing the RAID can change state. To accurately capture this behavior, the model becomes a Markov regenerative process (MRGP) [23]. In our MRGP model, the rebuild time is not necessarily exponentially distributed and hence it is a higher fidelity representation of RAID storage behavior compared to CTMC models.

The MRGP model is used to the performability comparison of RAID10 with RAID6 which are the two most popular enterprise RAID architectures. RAID10 is sometimes called a hierarchical RAID which combines mirroring (RAID1) and striping (RAID0). Due to its better write performance, generally RAID10 is preferred by users over RAID6. However, in terms of disk failure tolerance, RAID10 is inferior to RAID6 as it can fail by a particular combination of double disk failures while RAID6 tolerates any double disk failure. Depending on the number of disks used, RAID6 also can achieve better sequential read performance compared to RAID10 with the same number of disks [109]. To better characterize the performability of RAID10 and RAID6, we employ MRGP models with reward assignment, known as Markov regenerative reward model (MRRM). We assign reward based on our performance benchmark measurements, and the resulting MRRMs yield the expected performability of RAID storage systems as its steady-state solution. We also discuss the tradeoffs between RAID10 and RAID6 storage architectures.

The rest of the section is organized as follows. In Section 5.2.1.2, we review the related work for reliability and performance analysis of RAID storage systems. Section 5.2.1.3 shows the traditional CTMC-based modeling approach for storage system and clarifies the modeling errors criticized by researchers. To overcome the issue of memory-less property assumed in traditional models, we present MRGP models for storage systems with RAID level 6 and 10 in Section 5.2.1.4. Section 5.2.1.5 gives the results of numerical studies on the presented MRGP models and shows that the data availability is insensitive to memory-less property assumed in CTMC models. We also conduct the performability comparison between RAID6 with RAID10 by combining the analytical results and performance benchmarks on the real storage system. Section 5.2.1.6 summarizes the findings and gives our conclusions.

5.2.1.2. Related work

Performability is defined as a composite measure of reliability (or availability) and performance of degradable systems [13]. In the late ‘70s, Beaudry presented performance-related reliability measures to take into account the different performance levels of degradable systems [110]. Huslende considered performance reliability by assuming a

minimum performance threshold and presented a threshold-based performability measure [14]. Smith et al. evaluated the performability of multiprocessor system by complementary distribution of time-averaged accumulated performance measure [15]. Some other related papers about the performability study are summarized in [18]. In this paper, we adopt MRRM based performability analysis, which was first studied by Logothetis et al [16], since we need to capture the non-exponential nature of RAID rebuild times.

An approach for assessing the performability of RAID storage system is presented by Sun et al [111]. They construct a CTMC model which captures the expected behavior of a storage system and combine the model with performance benchmark results for performability assessment. A new performability metric called P-Graph is used to visualize performability of storage systems. Although their availability model is comprehensive as it captures failures of several components like RAID controller, all the state transition times are assumed to be exponentially distributed. Our performability analysis extends their work with non-exponential distribution of the rebuild times.

Thomasian et al. presented analytical studies on reliability and performance of storage systems with different RAID configurations [112][113][114]. Several different RAID1 organizations are compared with each other [112] and their performance is compared against RAID5 system [113]. While MTTDL has been used as a reliability measure in their papers, in a recent study, the authors compute the reliability, in terms of MTTDL, without considering repair operations [114]. Our performability study differs from their work as we compute the performability as a combination of availability and performance. We focus on a simple RAID10 configuration that is implemented by a popular RAID controller and evaluate the performability with real benchmark results.

It is well-known that the reliability of disk-based storage systems is highly influenced by latent sector errors (LSE), i.e., errors that go undetected until the corresponding disk sectors are accessed. A large-scale field study on LSE revealed the high degree of temporal locality between successive LSE occurrences [115]. The practical approach to cope with LSEs is disk scrubbing that continually scans the disk in order to detect LSEs proactively [116][117]. Intra-disk redundancy is proposed as another mechanism to protect against LSEs [118].

Comparative study of these two mechanisms is found in [119][120] and further comprehensive analysis and corrected arguments are presented by Iliadis et al [107]. The reliability model incorporating the impacts of LSEs is presented by Wu et al. [121] and the effectiveness of disk scrubbing is first studied by Schwarz et al [117]. Besides LSE, undetected disk errors (UDE) are another type of problem in storage systems. UDEs are silent data corruption events and have potential to corrupt data being delivered to user applications. Rozier et al. presented a reliability model of RAID system with UDE and presented a hybrid solution method which combines discrete event simulation and analytic-numerical solution [122]. In this paper, we do not incorporate the impacts of LSEs, scrubbing or UDEs in the models. Although it is a challenging future research to extend the MRGP model to incorporate these features. We believe that analytical solution might become a problem in this case and discrete-event simulation or hybrid solution approach as used in [101][122] will be more needed.

5.2.1.3. RAID Markov models

This section reviews the CTMC-based reliability models for RAID storage systems. Consider a disk array with N disks each of which can fail at a constant rate λ. When a disk fails, a rebuild operation is carried out to recover the data on a spare disk whose recovery rate is assumed to be a constant rate μ. We assume that the failed disk is replaced with a new spare disk so that the total number of operational disks within the RAID array does not change. The state transitions of RAID system can be represented by a CTMC in which states are labeled by the number of failed disks.

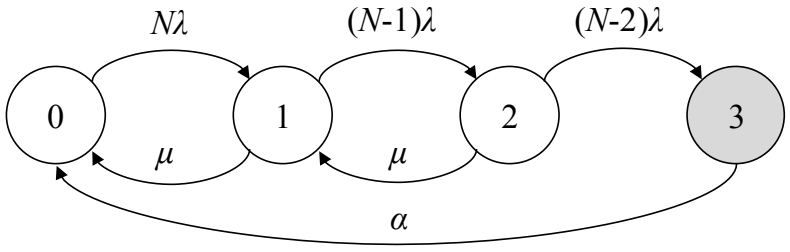


Figure 5.2 CTMC model for RAID6 storage system

Figure 5.2 shows an instance of CTMC model for RAID6 configuration where we further assume that the entire disk array should be replaced (at a constant rate α) after encountering triple disk failures. Note that in state 0 there are N available disks which results in λ times N being the state transition rate to state 1. State 1 and state 2 have similar state transition rates to state 2 and state 3 depending on the respective number of available disks (as presented by $(N-1)\lambda$ and $(N-2)\lambda$). The RAID6 CTMC model yields a closed form solution for steady-state availability that is the sum of steady-state probabilities of states 0, 1 and 2:

$$A_{\mathrm{RAID6}} = \sum_{i=0,1,2} \pi_i = \frac{\alpha\{2(N-1)^2\lambda^2 + (N\lambda+\mu)(\mu+(N-2)\lambda)\}}{\alpha\{2(N-1)^2\lambda^2 + (N\lambda+\mu)(\mu+(N-2)\lambda)\} + N(N-1)(N-2)\lambda^3}$$

Although the CTMC model provides a reasonably simple representation of state transitions of a RAID system, several drawbacks are discussed in the literature. The first criticism is directed at the unreality of the assumption of exponential distribution. Empirical studies on disk failure trends show that the time to disk failure does not fit exponential distribution and two parameter distributions such as Weibull are more suitable [100][101]. Since disk failures can occur due to different causes, it is unrealistic to model a single state transition with a constant failure rate. Failure distributions can be mixtures of multiple distributions because of several failure modes and distinct production vintages. The second criticism is related to the memory-less property of the sojourn times in a homogeneous CTMC that results in modeling errors. Under the memory-less assumption, the state transition rate does not depend on the amount of time spent in a state. The disk failure rate is not changed according to the age and all the intermediate results of rebuild process are cleared if another disk fails during rebuild operation. However, in reality, disk failure rate depends on its age and disk rebuild progress is not cleared by another disk failure.

In fact, these two issues are connected because memoryless property is equivalent to the time-independent transition rate (i.e., exponential distribution). In other words, the memoryless property holds only when we assume exponential distributions for all the state transitions. Extending the model by replacing disk failure distributions with more relevant ones is certainly possible. However, the difference in time to failure distribution might not to

have a huge impact on the steady-state measure as Venkatesan and Iliadis's study on the impact of failure distributions on MTTDL [106] shows. The assumption of exponential distributions for disk failure could be acceptable in the early phase of system design where users do not have any empirical data for disk failure times.

The memoryless assumption of disk rebuild time is more problematic. According to the common implementation of disk rebuild operation in RAID, the rebuild process is cumulative and the operation can continue execution even after another disk failure. The CTMC does not account for this non-exponential behavior. In contrast with the uncertainty of disk failure distributions, disk rebuild time can be controlled by the design and implementation. The modeling error caused by the limitation of CTMC is thus far not studied for RAID architecture. We investigate this issue with comprehensive Markov regenerative models.

5.2.1.4. RAID Markov regenerative models

We propose Markov regenerative process (MRGP) models for RAID storage systems in order to capture non-exponential nature of the disk rebuild times.

5.2.1.4.1. Definition of MRGP

Having its basis in Markov renewal theory, MRGP allows state transitions with general distribution in a state space model. An MRGP is a stochastic process $\{Z(t); t \geq 0\}$ with state space Φ which has regeneration time points at which the process probabilistically restarts itself. The stochastic process between regeneration epochs does not necessarily have Markov property, but the sequence of regeneration time points satisfy Markov property such that the future evolution of the stochastic process, given the process state at a regeneration point, does not depend on the history before that point. The definition of MRGP is provides in Section 3.2.6.

5.2.1.4.2. RAID6 MRGP model

In a RAID storage system which has the property of Double Disk Failure (DDF) tolerance, the rebuild operation of a failed disk continues even after another disk failure. The state of

the RAID system is changed with the remaining memory of the time elapsed since the beginning of the rebuild operation. Such time dependent behaviors across states can be modeled by an MRGP. Figure 5.3 shows the MRGP RAID model for RAID6 configuration where Z_{ij}, $i, j \in \Phi$ denotes the random variable for the transition time from state *i* to state *j*.

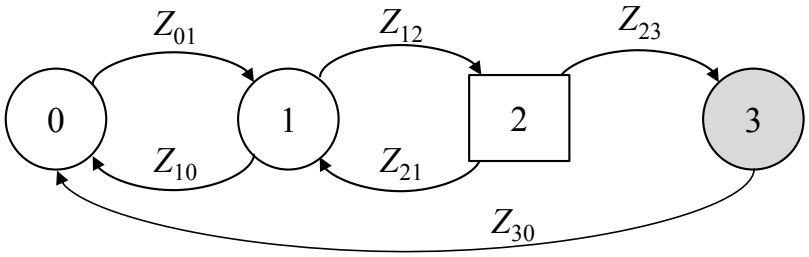


Figure 5.3 MRGP model for RAID6 storage system

A difference from the CTMC model (shown in Section 5.2.1.3) is the introduction of a state with time-dependent exit rates denoted by a rectangle (i.e., state 2) in which the process does not regenerate itself. In state 2, the distribution for the time to rebuild a failed disk depends on the time spent in state 1 and hence it is a non-regenerative state. By contrast, the process is regenerated at the entrance of all the other states 0, 1, or 3 which are represented by circles.

We assume that the time to disk failure follows an exponential distribution with rate λ whereas the time to rebuild a disk and reconstruct RAID system follows general distributions $G_1(t)$ and $G_2(t)$, respectively.

The sequence of visits to states in $\Omega=\{0,1,3\}$ form an embedded Markov chain of the MRGP. The global kernel of the RAID6 MRGP model is:

$$\mathbf{K}(t) = \begin{bmatrix} 0 & K_{01}(t) & 0 \\ K_{10}(t) & K_{11}(t) & K_{13}(t) \\ K_{30}(t) & 0 & 0 \end{bmatrix}.$$

Note that the subscripts *ij* on $K_{ij}(t)$ denote the actual state labels in $\Omega = \{0,1,3\}$ (and they are not the indices of the matrix). From the definition of kernel distributions, $K_{10}(t)$ is the conditional probability that the process has regenerated into state 0 by time *t* given the prior

regeneration occurred in state 1 at time 0. It is the probability that the rebuild operation for the failed disk completes before encountering another disk failure. Thus

$$K_{10}(t) = \Pr\{Z_{10} \le t, Z_{12} > Z_{10}\} = \int_0^t \left(1 - F_{Z_{12}}(x)\right) dF_{Z_{10}}(x) = \int_0^t e^{-(N-1)\lambda x} dG_1(x)$$

Next, $K_{11}(t)$ is derived as the probability that the process starting from state 1 has regenerated again into state 1 by time t after visiting state 2. Note that the process is not regenerated at the entrance of state 2 and holds the memory of sojourn time in state 1. A disk failure occurs while the rebuild operation is ongoing, but the rebuild operation does complete before a triple disk failure (in state 3). Let R be the random variable for time to rebuild a failed disk whose distribution is $G_1(t)$, and Z_{13}be the random variable representing the time to triple disk failures from a single disk failure. Note that Z_{13} is the sum of the two variables Z_{12} and Z_{23} in Figure 5.3. $K_{11}(t)$ is then given by:

$$\begin{aligned} K_{11}(t) &= \Pr\{R \le t, Z_{13} > R\} - \Pr\{R \le t, Z_{12} > R\} \\ &= \int_0^t \left(1 - F_{Z_{13}}(x)\right) dF_R(x) - \int_0^t \left(1 - F_{Z_{12}}(x)\right) dF_R(x) \\ &= \int_0^t \left\{(N-1)e^{-(N-2)\lambda x} - (N-2)e^{-(N-1)\lambda x}\right\} dG_1(x) - \int_0^t e^{-(N-1)\lambda x} dG_1(x) \\ &= (N-1)\int_0^t \left\{e^{-(N-2)\lambda x} - e^{-(N-1)\lambda x}\right\} dG_1(x)\,. \end{aligned}$$

In a similar fashion, $K_{13}(t)$ is obtained as:

$$\begin{aligned} K_{13}(t) &= \Pr\{Z_{13} \le t, R > Z_{13}\} = \int_0^t \left(1 - F_R(x)\right) dF_{Z_{13}}(x) \\ &= (N-1)(N-2)\lambda \int_0^t \left(1 - G_1(x)\right)\left(e^{-(N-2)\lambda x} - e^{-(N-1)\lambda x}\right) dx. \end{aligned}$$

Let $K_{10}(\infty)$, $K_{13}(\infty)$ and $K_{11}(\infty)$ be denoted by a, b and 1-a-b, respectively. Solving the linear system $v = v \cdot \mathbf{K}(\infty)$ with $\sum_{i \in \Omega} v_i = 1$, the steady-state probabilities of the embedded Markov chain are:

$$v_0 = \frac{a+b}{a+2b+1}, \quad v_1 = \frac{1}{a+2b+1}, \quad v_3 = \frac{b}{a+2b+1}.$$

Next, we derive the local kernel of the RAID6 MRGP model. The local kernel describes the process dynamics between two consecutive regeneration time points and hence is given by a 3×4 matrix as follows:

$$\mathbf{E}(t) = \begin{bmatrix} E_{00}(t) & 0 & 0 & 0 \\ 0 & E_{11}(t) & E_{12}(t) & 0 \\ 0 & 0 & 0 & E_{33}(t) \end{bmatrix}.$$

$E_{ii}(t)$, $i \in \{0,1,3\}$ represent the probabilities that the process starting in state i stays in the same state till time t. Therefore, we have:

$$E_{00}(t) = 1 - F_{Z_{01}}(t) = e^{-N\lambda t},$$

$$E_{11}(t) = \left(1 - F_{Z_{10}}(t)\right)\left(1 - F_{Z_{12}}(t)\right) = \left(1 - G_1(t)\right)e^{-(N-1)\lambda t},$$

$$E_{33}(t) = 1 - F_{Z_{30}}(t) = 1 - G_2(t).$$

Meanwhile, $E_{12}(t)$ is the probability that the process starting from state 1 is in state 2 at time t:

$$\begin{aligned} E_{12}(t) &= \Pr\{R > t, Z_{13} > t \geq Z_{12}\} \\ &= \Pr\{R > t, Z_{13} > t\} - \Pr\{R > t, Z_{12} > t\} \\ &= \left(1 - F_R(t)\right)\left(1 - F_{Z_{13}}(t)\right) - \left(1 - F_R(t)\right)\left(1 - F_{Z_{12}}(t)\right) \\ &= \left(1 - F_R(t)\right)\left(F_{Z_{12}}(t) - F_{Z_{13}}(t)\right) \\ &= \left(1 - G_1(t)\right)(N-1)\left(e^{-(N-2)\lambda t} - e^{-(N-1)\lambda t}\right) \end{aligned}$$

From the local kernel distributions, the mean sojourn times are computed as:

$$\alpha_{00} = 1/N\lambda,$$

$$\alpha_{11} = \int_0^\infty \left(1 - G_1(t)\right)e^{-(N-1)\lambda t}dt,$$

$$\alpha_{33} = \int_0^\infty \big(1 - G_2(t)\big)dt,$$

$$\alpha_{12} = (N-1)\int_0^\infty \big(1 - G_1(t)\big)\big(e^{-(N-2)\lambda t} - e^{-(N-1)\lambda t}\big)dt.$$

The data on the storage system is accessible when the storage system is in state 0, 1, or 2. Hence the expected data availability is computed by:

$$A_{RAID6} = \sum_{i=0,1,2} \pi_i = \sum_{i=0,1,2} \frac{\sum_{k\in\Omega} v_k \alpha_{ki}}{\sum_{k\in\Omega} v_k \sum_{l\in\Omega} \alpha_{kl}} = \frac{(a+b)\alpha_{00} + \alpha_{11} + \alpha_{12}}{(a+b)\alpha_{00} + \alpha_{11} + \alpha_{12} + b\alpha_{33}}. \quad (25)$$

and the performability as:

$$P_{RAID6} = \sum_{j\in\Phi} \pi_j \cdot r_j = \frac{(a+b)\alpha_{00}r_0 + \alpha_{11}r_1 + \alpha_{12}r_2}{(a+b)\alpha_{00} + \alpha_{11} + \alpha_{12} + b\alpha_{33}}. \quad (26)$$

Since A_{RAID6} has the following relationship with the traditional notion of mean time to data loss

$$A_{\mathrm{RAID6}} = \frac{MTTDL_{\mathrm{RAID6}}}{MTTDL_{\mathrm{RAID6}} + \alpha_{33}},$$

$MTTDL_{\mathrm{RAID6}}$ is computed by

$$MTTDL_{RAID6} = \frac{A_{RAID6} \cdot \alpha_{33}}{1 - A_{RAID6}} = \frac{(a+b)\alpha_{00} + \alpha_{11} + \alpha_{12}}{b}. \quad (27)$$

Now if we assume that the rebuild and reconstruction times are deterministic and the distribution functions are given by $G_1(t) = \mathrm{u}(t-\tau_1)$ and $G_2(t) = \mathrm{u}(t-\tau_2)$ where u(·) is the unit step function, we have:

$$a = e^{-(N-1)\lambda\tau_1},$$

$$b = 1 - (N-1)e^{-(N-2)\lambda\tau_1} - (N-2)e^{-(N-1)\lambda\tau_1},$$

$$\alpha_{11} = \frac{1}{(N-1)\lambda}\big[1 - e^{-(N-1)\lambda\tau_1}\big],$$

$$\alpha_{33} = \tau_2,$$

$$\alpha_{12} = \frac{1 - (N-1)e^{-(N-2)\lambda\tau_1} + (N-2)e^{-(N-1)\lambda\tau_1}}{(N-2)\lambda}.$$

5.2.1.4.3. RAID10 MRGP model

We construct an MRGP model for RAID10 configuration with *N* disks ($N \geq 6$). Like the RAID6 MRGP model, the non-exponentially distributed rebuild time at another disk failure event can be modeled by introducing states with time-dependent exit rates. Since the probability of more than four disk failures is negligibly small, we neglect the storage failures caused by more than four disk failures. With this approximation, the RAID10 MRGP model is shown in Figure 5.4.

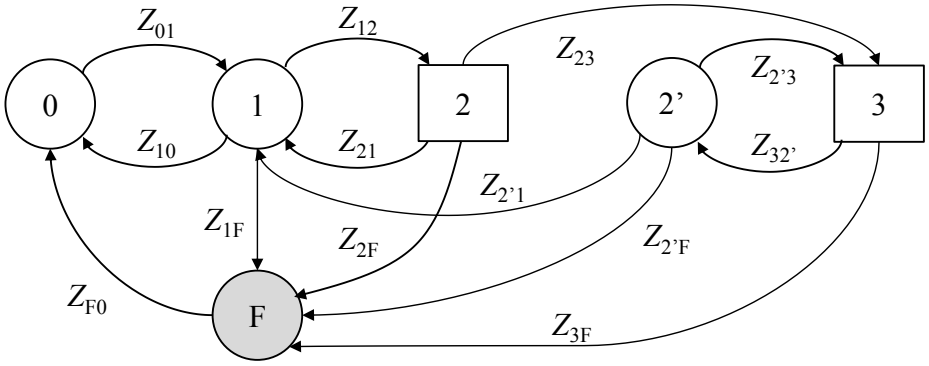


Figure 5.4 MRGP model for RAID10 storage system

The state labels 0, 1, 2, 3 represent the number of failed disks in the storage system, whereas state F is the storage failure state. A RAID10 storage system can fail even by two disk failures depending on the combination of failed disks. If the second failure occurs at the mirroring pair of the first failed disk, the storage loses data (the model enters state F). When the system is in state 1, another disk failure causes storage failure with probability one divided by the number of working disks (i.e., $1/(N-1)$). The storage state in which two disks have failed is divided into two states; state 2 caused by a disk failure during the rebuild operation to the prior failed disk and state 2' resulting from the completion of the rebuild operation to one of the three failed disks. The stochastic process is not regenerated at the entrance of state 2, while it regenerates at the entrance of state 2'. From state 2 or state 2', the storage system can fail with probability two divided by the number of working disks (i.e., $2/(N-2)$). There is a risk that a third disk fails before completing a rebuild operation to the first failed disk, resulting in state 3. Since the rebuild operation continues its execution even

after entering into state 3, the latter is a non-regenerative state. The embedded Markov chain is formed by the sequence of state transitions among the regenerative states $\Omega = \{0,1,2',\mathrm{F}\}$.

We assume disk failure time is exponentially distributed with rate λ and define $G_1(t)$ and $G_2(t)$ as the distribution functions for disk rebuild time and RAID reconstruction time, respectively. The global kernel of the RAID10 MRGP model is given by:

$$\mathbf{K}(t) = \begin{bmatrix} 0 & K_{01}(t) & 0 & 0 \\ K_{10}(t) & K_{11}(t) & K_{12'}(t) & K_{1\mathrm{F}}(t) \\ 0 & K_{2'1}(t) & K_{2'2'}(t) & K_{2'\mathrm{F}}(t) \\ K_{\mathrm{F}0}(t) & 0 & 0 & 0 \end{bmatrix}.$$

$K_{10}(t)$ is the conditional probability that the process has regenerated into state 0 by time t given the prior regeneration occurred in state 1 at time zero. This corresponds to the event that a rebuild process completes before encountering another disk failure. Thus

$$\begin{aligned} K_{10}(t) &= \Pr\{Z_{10} \le t, Z_{1\mathrm{F}} > Z_{10}, Z_{12} > Z_{10}\} \\ &= \int_0^t \left(1 - F_{Z_{1\mathrm{F}}}(t)\right)\left(1 - F_{Z_{12}}(t)\right) dF_{Z_{10}}(x) \\ &= \int_0^t e^{-(N-1)\lambda x} dG_1(x). \end{aligned}$$

Next, $K_{11}(t)$ is the probability that the process starting from state 1 has regenerated again into state 1 by time t after visiting state 2. Conditioning on $Z_{12} = \delta < Z_{10}, Z_{1\mathrm{F}}$, we have

$$\begin{aligned} &\Pr\{Z_{12} + Z_{21} \le t, Z_{21} < Z_{2\mathrm{F}}, Z_{21} < Z_{23} | Z_{12} = \delta < Z_{10}, Z_{1\mathrm{F}}\} \\ &\quad = \Pr\{Z_{21} \le t - \delta, Z_{21} < Z_{2\mathrm{F}}, Z_{21} < Z_{23} | Z_{12} = \delta < Z_{10}, Z_{1\mathrm{F}}\} \\ &\quad = \int_\delta^t e^{-(N-2)\lambda(x-\delta)} dG_1(x | x > \delta). \end{aligned}$$

Unconditioning on Z_{12}, $K_{11}(t)$ is given by:

$$\begin{aligned} K_{11}(t) &= \Pr\{Z_{12} + Z_{21} \le t, Z_{21} < Z_{2\mathrm{F}}, Z_{21} < Z_{23}, Z_{12} < Z_{10}, Z_{12} < Z_{1\mathrm{F}}\} \\ &= \int_0^t \Pr\{Z_{21} \le t - \delta, Z_{21} < Z_{2\mathrm{F}}, Z_{23} | Z_{12} = \delta < Z_{10}, Z_{1\mathrm{F}}\} \\ &\qquad \cdot \left(1 - F_{Z_{10}}(\delta)\right)\left(1 - F_{Z_{1\mathrm{F}}}(\delta)\right) dF_{Z_{12}}(\delta) \end{aligned}$$

$$= \int_0^t \left\{ \int_\delta^t e^{-(N-2)\lambda(x-\delta)} dG_1(x|x>\delta) \right\} \cdot \left(1 - G_1(\delta)\right) \cdot (N-2)\lambda e^{-(N-1)\lambda\delta}\, d\delta.$$

In a similar manner, we derive $K_{12'}(t)$ by first considering the probability conditioned on the time to reach state 3 from state 1 through state 2, $Z_{12} + Z_{23} = \delta_2$,

$$\Pr\left\{ \begin{matrix} Z_{12} + Z_{23} + Z_{32'} \le t, Z_{32'} < Z_{3\mathrm{F}} | Z_{12} + Z_{23} = \delta_2 \\ , Z_{12} < Z_{10}, Z_{12} < Z_{1\mathrm{F}}, Z_{23} < Z_{21}, Z_{23} < Z_{2\mathrm{F}} \end{matrix} \right\}$$

$$= \Pr\left\{ \begin{matrix} Z_{32'} \le t - \delta_2, Z_{32'} < Z_{3\mathrm{F}} | Z_{12} + Z_{23} = \delta_2 \\ , Z_{12} < Z_{10}, Z_{12} < Z_{1\mathrm{F}}, Z_{23} < Z_{21}, Z_{23} < Z_{2\mathrm{F}} \end{matrix} \right\}$$

$$= \int_{\delta_2}^t e^{-3\lambda(x-\delta_2)} dG_1(x|x>\delta_2).$$

Further conditioning on $Z_{12} = \delta_1 < Z_{10}, Z_{1\mathrm{F}}$ while unconditioning on $Z_{12} + Z_{23} = \delta_2$,

$$\Pr\{Z_{12} + Z_{23} + Z_{32'} \le t, Z_{23} < Z_{21}, Z_{23} < Z_{2\mathrm{F}}, Z_{32'} < Z_{3\mathrm{F}} | Z_{12} = \delta_1 < Z_{10}, Z_{1\mathrm{F}}\}$$

$$= \int_{\delta_1}^t \Pr\left\{ \begin{matrix} Z_{12} + Z_{23} + Z_{32'} \le t, Z_{32'} < Z_{3\mathrm{F}} | Z_{12} + Z_{23} = \delta_2 \\ , Z_{12} < Z_{10}, Z_{12} < Z_{1\mathrm{F}}, Z_{23} < Z_{21}, Z_{23} < Z_{2\mathrm{F}} \end{matrix} \right\} \cdot$$

$$\left(1 - F_{Z_{21}}(\delta_2 - \delta_1 | Z_{12} = \delta_1 < Z_{10}, Z_{1\mathrm{F}})\right) \cdot \left(1 - F_{Z_{2\mathrm{F}}}(\delta_2 - \delta_1 | Z_{12} = \delta_1 < Z_{10}, Z_{1\mathrm{F}})\right)$$

$$\cdot\, dF_{Z_{123}}(\delta_2 | Z_{12} = \delta_1 < Z_{10}, Z_{1\mathrm{F}})$$

$$= \int_{\delta_1}^t \left\{ \int_{\delta_2}^t e^{-3\lambda(x-\delta_2)} dG_1(x|x>\delta_2) \right\} \cdot \left(1 - G_1(\delta_2|\delta_2 > \delta_1)\right) \cdot e^{-2\lambda(\delta_2 - \delta_1)}$$

$$\cdot\, (N-4)\lambda e^{-(N-4)\lambda(\delta_2-\delta_1)} d\delta_2.$$

Unconditioning on Z_{12}, we obtain

$$K_{12'}(t) = \Pr\left\{ \begin{matrix} Z_{12} + Z_{23} + Z_{32'} \le t, Z_{12} < Z_{10}, Z_{12} < Z_{1\mathrm{F}} \\ , Z_{23} < Z_{21}, Z_{23} < Z_{2\mathrm{F}}, Z_{32'} < Z_{3\mathrm{F}} \end{matrix} \right\}$$

$$= (N-2)(N-4)\lambda^2 \int_0^t \left\{ \int_{\delta_1}^t \left\{ \int_{\delta_2}^t e^{-3\lambda(x-\delta_2)} dG_1(x|x>\delta_2) \right\} \cdot \left(1 - G_1(\delta_2|\delta_2 > \delta_1)\right) \right.$$

$$\left. \cdot\, e^{-(N-2)\lambda(\delta_2-\delta_1)} d\delta_2 \right\} \cdot \left(1 - G_1(\delta_1)\right) \cdot e^{-(N-1)\lambda\delta_1}\, d\delta_1$$

Considering the rebuild operation started from state 2', $K_{2'1}(t)$ is given by:

$$K_{2'1}(t) = \Pr\{Z_{2'1} \le t, Z_{2'1} < Z_{2'3}, Z_{2'1} < Z_{2'\mathrm{F}}\}$$

$$= \int_0^t \left(1 - F_{Z_{2'3}}(x)\right)\left(1 - F_{Z_{2'F}}(x)\right) dF_{Z_{2'1}}(x)$$

$$= \int_0^t e^{-(N-2)\lambda x} dG_1(x).$$

$K_{2'2'}(t)$ is the probability that the process starting from state 2' regenerates again into state 2' at time t after visiting state 3. By conditioning on $Z_{2'3} = \delta < Z_{2'1}, Z_{2'F}$,

$$\Pr\{Z_{2'3} + Z_{32'} \le t, Z_{32'} < Z_{3F} | Z_{2'3} = \delta < Z_{2'1}, Z_{2'F}\}$$
$$= \Pr\{Z_{32'} \le t - \delta, Z_{32'} < Z_{3F} | Z_{2'3} = \delta < Z_{2'1}, Z_{2'F}\}$$
$$= \int_\delta^t e^{-3\lambda(x-\delta)} dG_1(x | x > \delta)$$

Unconditioning on $Z_{2'3}$, $K_{2'2'}(t)$ is:

$$K_{2'2'}(t) = \Pr\{Z_{2'3} + Z_{32'} \le t, Z_{2'3} < Z_{2'1}, Z_{2'3} < Z_{2'F}\}$$

$$= \int_0^t \left\{ \int_\delta^t e^{-3\lambda(x-\delta)} dG_1(x | x > \delta) \right\} \cdot \left(1 - G_1(\delta)\right) \cdot (N-4)\lambda e^{-(N-2)\lambda\delta}\, d\delta.$$

Let $K_{10}(\infty), K_{11}(\infty), K_{12'}(\infty), K_{2'1}(\infty)$ and $K_{2'2'}(\infty)$ be denoted by a, b, c, d and e, respectively. The transition probability matrix of the embedded Markov chain is then given by

$$\mathbf{K}(\infty) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ a & b & c & 1-(a+b+c) \\ 0 & d & e & 1-(d+e) \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Solving the linear system $v = v \cdot \mathbf{K}(\infty)$ with $\sum_{i \in \Omega} v_i = 1$, the steady-state probabilities of the embedded Markov chain are:

$$(v_0, v_1, v_{2'}, v_F) = \left(\frac{1 - b - \frac{cd}{1-e}}{w}, \quad \frac{1}{w}, \quad \frac{\frac{c}{1-e}}{w}, \quad \frac{1 - a - b - \frac{c}{1-e}}{w} \right),$$

where

$$w = 3 + 2b - a + \frac{d}{1-e}(1 - 2c).$$

Next, the local kernel of RAID10 MRGP model is derived as a 4×6 matrix

$$\mathbf{E}(t) = \begin{bmatrix} E_{00}(t) & 0 & 0 & 0 & 0 & 0 \\ 0 & E_{11}(t) & E_{12}(t) & E_{13}(t) & 0 & 0 \\ 0 & 0 & 0 & E_{2'3}(t) & E_{2'2'}(t) & 0 \\ 0 & 0 & 0 & 0 & 0 & E_{\mathrm{FF}}(t) \end{bmatrix}.$$

$E_{ii}(t)$, $i \in \{0,1,2',\mathrm{F}\}$ are given by

$$E_{00}(t) = 1 - F_{Z_{01}}(t) = e^{-N\lambda t},$$

$$E_{11}(t) = \Pr\{Z_{10} > t, Z_{1\mathrm{F}} > t, Z_{12} > t\} = \big(1 - G_1(t)\big)e^{-(N-1)\lambda t},$$

$$E_{2'2'}(t) = \Pr\{Z_{2'1} > t, Z_{2'F} > t, Z_{2'3} > t\} = \big(1 - G_1(t)\big)e^{-(N-2)\lambda t},$$

$$E_{\mathrm{FF}}(t) = 1 - F_{Z_{\mathrm{F0}}}(t) = 1 - G_2(t).$$

$E_{12}(t)$ is the probability that the process starting from state 1 is in state 2 at time t. To capture this probability, we consider the failure of disk D, which is the mirror of the failed disk in state 1, separately from other disk failures. Let $Z_{1\mathrm{F}}{}^{*}$ denote the random variable for the time to failure of disk D. Such a failure results in RAID entering the state F, either directly from state 1 or through state 2 or state 3. Let R and U be the random variables for the time to rebuild a failed disk and the time to two more failures of disks other than D (either state 3 or state F via state 2) from state 1, respectively. $E_{12}(t)$ is then given by

$$\begin{aligned} E_{12}(t) &= \Pr\{R > t, Z_{1\mathrm{F}}{}^{*} > t, U > t \geq Z_{12}\} \\ &= \Pr\{R > t, Z_{1\mathrm{F}}{}^{*} > t, U > t\} - \Pr\{R > t, Z_{1\mathrm{F}}{}^{*} > t, Z_{12} > t\} \\ &= \big(1 - G_1(t)\big) \cdot e^{-\lambda t} \cdot \Big(F_{Z_{12}}(t) - F_U(t)\Big). \end{aligned}$$

The failure rate to state 2 from state 1 is equal to $(N-2)\lambda$, since individual disk failure time is exponentially distributed with rate λ and a failure of the mirrored disk paired with the previously failed disk needs to be excluded from this case (it is accounted for in the transition from state 2 to state F). Similarly, the failure rate to state 3 from state 2 is given by $(N-3)\lambda$. Therefore, $F_U(t)$ follows a 2-stage hypo-exponential distribution with parameters $(N-2)\lambda$ and $(N-3)\lambda$:

$$F_U(t) = 1 - (N-2)e^{-(N-3)\lambda t} + (N-3)e^{-(N-2)\lambda t}.$$

Thus, $E_{12}(t)$ can be obtained as

$$E_{12}(t) = (N-2)\big(1-G_1(t)\big)\big(e^{-(N-2)\lambda t} - e^{-(N-1)\lambda t}\big).$$

Next, $E_{13}(t)$ is the probability that the process starting from state 1 stays in state 3 at time t. Similar to the case of $E_{12}(t)$, we separate the failures of the disk D and D_2, which are disks paired with the ones failed at the entrance of state 1 and state 2 respectively, from other disk failures. Let Z_{2F}^* and Z_{23F}^* be the random variables for the time to fail the paired disk D_2 resulting in RAID failure from state 2 and the time to RAID failure due to quadruple disk failures caused by disks other than D and D_2 from state 2. Note that Z_{2F}^* is exponentially distributed with parameter λ and Z_{23F}^* follows two-stage hypo-exponential distribution with parameter $(N-4)\lambda$, which does not include a failure of mirrored disk, and λ that corresponds to the transition from state 3 to state F. $E_{13}(t)$ is given by

$$\begin{aligned}
E_{13}(t) &= \Pr\{R > t, Z_{1F}^* > t, Z_{12} + Z_{2F}^* > t, Z_{12} + Z_{23F}^* > t \geq Z_{123}\} \\
&= \Pr\{R > t, Z_{1F}^* > t, Z_{12} + Z_{2F}^* > t, Z_{12} + Z_{23F}^* > t\} \\
&\qquad - \Pr\{R > t, Z_{1F}^* > t, Z_{12} + Z_{2F}^* > t, Z_{12} + Z_{23} > t\} \\
&= \big(1-G_1(t)\big)e^{-\lambda t}\int_{\delta=0}^{t}\Big(1-F_{Z_{2F}^*}(t-\delta)\Big)\cdot\Big(F_{Z_{23}}(t-\delta) - F_{Z_{23F}^*}(t-\delta)\Big)\,dF_{Z_{12}}(\delta) \\
&= \big(1-G_1(t)\big)e^{-\lambda t}\int_{\delta=0}^{t}\Big(1-F_{Z_{2F}^*}(t-\delta)\Big)\cdot\Big(F_{Z_{23}}(t-\delta) - F_{Z_{23F}^*}(t-\delta)\Big)(N \\
&\qquad - 2)\lambda e^{-(N-2)\lambda\delta}d\delta \\
&= \big(1-G_1(t)\big)(N-2)\lambda e^{-(N-1)\lambda t}\int_{x=0}^{t}\Big(1-F_{Z_{2F}^*}(x)\Big) \\
&\qquad \cdot\Big(F_{Z_{23}}(x) - F_{Z_{23F}^*}(x)\Big)e^{(N-2)\lambda x}\,dx \\
&= \frac{N-2}{2(N-5)}\big(1-G_1(t)\big)\big(2e^{-3\lambda t} + (N-6)e^{-(N-1)\lambda t} - (N-4)e^{-(N-3)\lambda t}\big).
\end{aligned}$$

$E_{2'3}(t)$ is the probability that the process starting from state 2' is in state 3 at time t. Let $Z_{2'F}^*$ and $Z_{2'3F}^*$ be the random variables for the time to RAID failure from state 2' due to disk failures whose mirrors are failed in state 2' and newly failed in state 3, respectively. $Z_{2'F}^*$ is exponentially distributed with parameter 2λ and Z_{23F}^* follows a two-stage hypo-exponential distribution with parameter $(N-4)\lambda$, and λ. $E_{2'3}(t)$ is then given by

$$
\begin{aligned}
E_{2'3}(t) &= \Pr\{R > t, Z_{2'\mathrm{F}} > t, Z_{2'3\mathrm{F}}{}^{*} > t \geq Z_{2'3}\} \\
&= \Pr\{R > t, Z_{2'\mathrm{F}}{}^{*} > t, Z_{2'3\mathrm{F}}{}^{*} > t\} - \Pr\{R > t, Z_{2'\mathrm{F}}{}^{*} > t, Z_{2'3} > t\} \\
&= \big(1 - G_1(t)\big) \cdot e^{-2\lambda t} \cdot \Big(F_{Z_{2'3}}(t) - F_{Z_{2'3\mathrm{F}}{}^{*}}(t)\Big) \\
&= \frac{N-4}{N-5}\big(1 - G_1(t)\big)\big(e^{-3\lambda t} - e^{-(N-2)\lambda t}\big).
\end{aligned}
$$

From the local kernel distributions, the mean sojourn times are obtained as:

$$\alpha_{00} = 1/N\lambda,$$

$$\alpha_{11} = \int_0^\infty \big(1 - G_1(t)\big)e^{-(N-1)\lambda t}dt,$$

$$\alpha_{2'2'} = \int_0^\infty \big(1 - G_1(t)\big)e^{-(N-2)\lambda t}dt,$$

$$\alpha_{\mathrm{FF}} = \int_0^\infty \big(1 - G_2(t)\big)dt,$$

$$\alpha_{12} = (N-2)\int_0^\infty \big(1 - G_1(t)\big)\big(e^{-(N-2)\lambda t} - e^{-(N-1)\lambda t}\big)dt,$$

$$\alpha_{13} = \frac{N-2}{2(N-5)}\int_0^\infty \big(1 - G_1(t)\big) \cdot \big(2e^{-3\lambda t} + (N-6)e^{-(N-1)\lambda t} - (N-4)e^{-(N-3)\lambda t}\big)dt.$$

Since the storage system is available in state 0, 1, 2, 2’ or 3, the expected data availability is computed by

$$
\begin{aligned}
A_{RAID10} &= \sum_{i=0,1,2,2',3} \pi_i = \sum_{i=0,1,2,2',3} \frac{\sum_{k\in\Omega} v_k \alpha_{ki}}{\sum_{k\in\Omega} v_k \sum_{l\in\Omega} \alpha_{kl}} \\
&= \frac{v_0\alpha_{00} + v_1(\alpha_{11} + \alpha_{12} + \alpha_{13}) + v_{2'}(\alpha_{2'2'} + \alpha_{2'3})}{v_0\alpha_{00} + v_1(\alpha_{11} + \alpha_{12} + \alpha_{13}) + v_{2'}(\alpha_{2'2'} + \alpha_{2'3}) + v_F\alpha_{FF}}.
\end{aligned}
\tag{28}
$$

Assigning reward rates to each state, the performability is

$$
\begin{aligned}
P_{RAID10} &= \sum_{j\in\Phi} \pi_j \cdot r_j \\
&= \frac{v_0\alpha_{00}r_0 + v_1\alpha_{11}r_1 + (v_1\alpha_{12} + v_{2'}\alpha_{2'2'})r_2 + (v_1\alpha_{13} + v_{2'}\alpha_{2'3})r_3}{v_0\alpha_{00} + v_1(\alpha_{11} + \alpha_{12} + \alpha_{13}) + v_{2'}(\alpha_{2'2'} + \alpha_{2'3}) + v_F\alpha_{FF}}.
\end{aligned}
\tag{29}
$$

From the availability, the mean time to data loss can be obtained as

$$MTTDL_{RAID10} = \frac{A_{RAID10} \cdot \alpha_{FF}}{1 - A_{RAID10}} = \frac{v_0\alpha_{00} + v_1(\alpha_{11} + \alpha_{12} + \alpha_{13}) + v_{2'}(\alpha_{2'2'} + \alpha_{2'3})}{v_F}. \quad (30)$$

When we assume deterministic times for disk rebuild and RAID reconstruction processes, the transition probabilities and the mean sojourn times are rewritten using $G_1(t) = \mathrm{u}(t - \tau_1)$ and $G_2(t) = \mathrm{u}(t - \tau_2)$:

$$a = e^{-(N-1)\lambda\tau_1},$$

$$b = (N-2)e^{-(N-2)\lambda\tau_1}\left(1 - e^{-\lambda\tau_1}\right),$$

$$c = \frac{(N-2)(N-4)}{N-5}\left\{\frac{e^{-3\lambda\tau_1}}{N-4}\left(1 - e^{-(N-4)\lambda\tau_1}\right) - e^{-(N-2)\lambda\tau_1}\left(1 - e^{-\lambda\tau_1}\right)\right\},$$

$$d = e^{-(N-2)\lambda\tau_1},$$

$$e = \frac{N-4}{N-5}e^{-3\lambda\tau_1}\left(1 - e^{-(N-5)\lambda\tau_1}\right),$$

$$\alpha_{11} = \frac{1}{(N-1)\lambda}\left(1 - e^{-(N-1)\lambda\tau_1}\right),$$

$$\alpha_{12} = \frac{1}{\lambda}\left(1 - e^{-(N-2)\lambda\tau_1}\right) - \frac{(N-2)}{(N-1)\lambda}\left(1 - e^{-(N-1)\lambda\tau_1}\right),$$

$$\alpha_{13} = \frac{N-2}{2(N-5)\lambda}\left\{\frac{2}{3}\cdot\left(1 - e^{-3\lambda\tau_1}\right) + \frac{N-6}{N-1}\cdot\left(1 - e^{-(N-1)\lambda\tau_1}\right) - \frac{N-4}{N-3} \cdot\left(1 - e^{-(N-3)\lambda\tau_1}\right)\right\},$$

$$\alpha_{2'2'} = \frac{1}{(N-2)\lambda}\left(1 - e^{-(N-2)\lambda\tau_1}\right),$$

$$\alpha_{2'3} = \frac{N-4}{(N-5)\lambda}\left\{\frac{1}{3}\left(1 - e^{-3\lambda\tau_1}\right) - \frac{1}{(N-2)}\left(1 - e^{-(N-2)\lambda\tau_1}\right)\right\},$$

$$\alpha_{\mathrm{FF}} = \tau_2.$$

Alternatively, if we assume the disk rebuild time and the RAID reconstruction times are exponentially distributed with rate α and μ, respectively, the MRGP model becomes CTMC and the memoryless property holds. Substituting $G_1(t) = 1 - e^{-\mu t}$ and $G_2(t) = 1 - e^{-\alpha t}$

yields the fully symbolic expressions for availability and performability for RAID10 storage system, which are consistent with those derived in the previous study [109].

5.2.1.5. Numerical results

In this section, we present the results of our numerical study on the proposed MRGP models. First, storage reliability is evaluated by the mean years to RAID storage failure using our MRGP model. Next, we focus on data availability perspectives where we compare the results of CTMC model and MRGP model and show that the difference between the estimated downtimes computed by CTMC and that by MRGP is negligibly small. Moreover, we compare the performability of RAID6 and RAID10 storage systems based on the proposed models with storage benchmark results. Finally, we conduct the sensitivity analysis to rebuild time distribution using gamma distribution with different parameter values.

5.2.1.5.1. RAID storage reliability

Using our MRGP models, first we compute the mean years to RAID storage failure as the reliability measure from (27) and (30). The parameter values used are shown in Table 5.1.

Table 5.1 Parameter values [99]

Parameters	**Values**	**Description**
N	6	Number of disks in the array
$1/\lambda$	10^4– 10^6 [hours]	Mean time to disk failure
$1/\mu$ ($=\tau_1$)	1 –24 [hours]	Mean time to disk rebuild
$1/\alpha(=\tau_2)$	24 [hours]	Mean time to storage reconstruction

The number of disks N is set to six because our experimental storage system used in the performability study consists of six disks. We believe the mean time to individual disk failure ranges from 10^4 hours to 10^6 hours according to specifications provided by disk vendors and experience of bad batches by users. Earlier studies of disk failure statistics also support this range of values [100] [123]. The disk rebuild time observed in our test system varies from

an hour to a few hours. Considering the rebuild time might prolong due to workload conditions and/or data volume, we vary the values in the range from one hour to 24 hours. When a storage failure occurs, we need manual operation to reconstruct the storage system and recover the data from backup. We assume it takes one day for these manual operations.

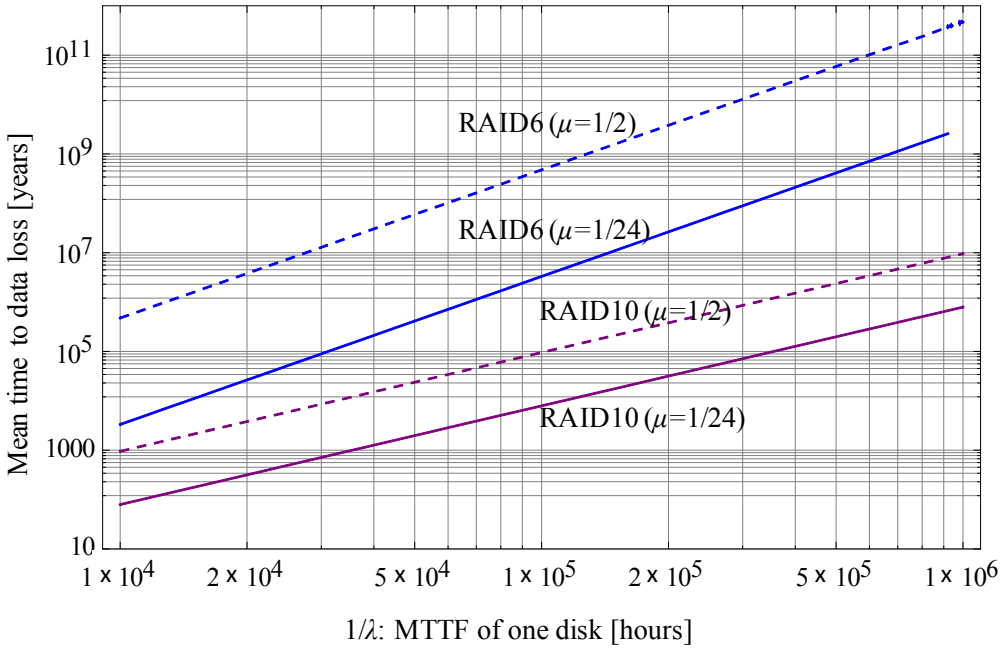


Figure 5.5 RAID storage reliability by years to storage failure [99]

Figure 5.5 shows the comparison of RAID storage reliabilities by varying the mean time to disk failure ($1/\lambda$) in the range [10^4– 10^6] with different disk rebuild rates (1/2 or 1/24). In the model, we assume deterministic transitions for disk rebuild time (two hours or 24 hours). As can be seen, the RAID6 architecture achieves higher reliability compared to the RAID10 architecture with the same number of disks, regardless of the MTTF of a single disk. Since the computation of mean time to storage failure uses our MRGP model, the non-exponential distributions of disk rebuild times are taken into account. The derivation of MTTDL under general rebuild time is studied in [106][124]. Although our derivation process is different from theirs, we confirm that our finding about the impact of variance on MTTDL agrees with their result through a sensitivity study using gamma distribution in the later section.

5.2.1.5.2. Data availability

Provided that the data is restorable from backup once the storage fails with data loss, data availability becomes a more concerned dependability measure that tells us how long the user cannot access the data on the storage. We compute the downtime from the data availability point of view for RAID6 and RAID10 storage systems using (25) and (28). Table 5.2 shows the downtime in seconds per year computed by the RAID6 and RAID10 MRGP models and the downtime of a disk array with no redundancy (i.e., RAID0 with six disks).

Table 5.2 Data Availability comparison: RAID6 vs. RAID10 [99]

$1/\mu$ [hours]	**$1/\lambda$[hours]**	**Downtime per year [seconds]**		
		RAID6	**RAID10**	**RAID0**
2	10^4	0.18150	90.8143	447671.9
	10^5	0.00018	0.90823	45346.54
	10^6	$<10^{-6}$	0.00908	4540.53
24	10^4	25.9041	1088.40	
	10^5	0.02613	10.8975	
	10^6	0.00003	0.10899	

The estimated downtime of RAID6 storage system is several orders of magnitude smaller than that of RAID10 regardless of the disk failure time and rebuild time. The results stem from the fact that RAID10 can fail with two disk failures, and imply that given the same number of disks, RAID6 configuration is preferable in terms of data availability.

Next, we evaluate the impact of memoryless assumption of rebuild operation times on the downtime through sensitivity analyses. First, we fix the mean time to disk rebuild to two hours and vary the mean time to disk failure from ten thousand hours to a million hours. Figure 5.6 shows the computed downtime from CTMC and MRGP models of RAID6 and RAID10.

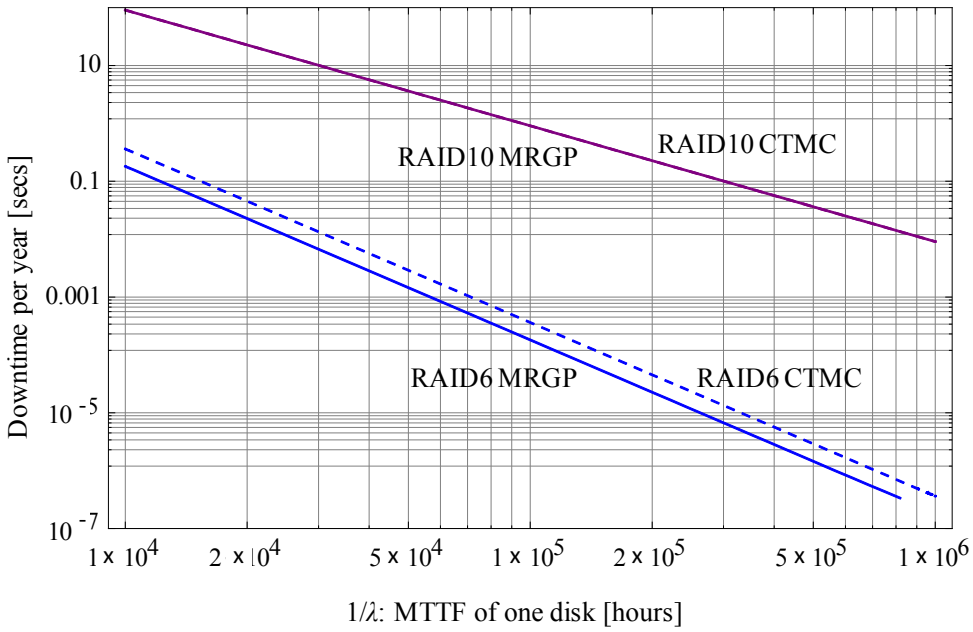


Figure 5.6 Comparison of downtime computed by CMTC and MRGP varying MTTF of a single disk [99]

As can be seen in RAID6 results, CTMC tends to overestimate the downtime due to the memoryless assumption of rebuild times which erroneously increases the downtime. However, the difference is negligibly small (<0.2 seconds) especially in the practical range of real disk failure rates (i.e., less than 10^{-4}). For RAID10, although we plotted both the results of MRGP and CTMC, the difference between two curves is too small to be visible on the graph. The difference between the results from CTMC and MRGP is less than 10^{-9} in the whole range of the sensitivity results.

Next, we look into the sensitivity of the downtime to mean rebuild times by fixing the disk failure rate to 10^{-6}[1/hours]. Figure 5.7 shows the comparison results obtained from CTMC and MRGP models for RAID6 and RAID10. We see that the CTMC overestimates the downtime regardless of disk rebuild times. Although the difference becomes larger as the disk rebuild time increases, it is marginal (<0.1 second) in both cases.

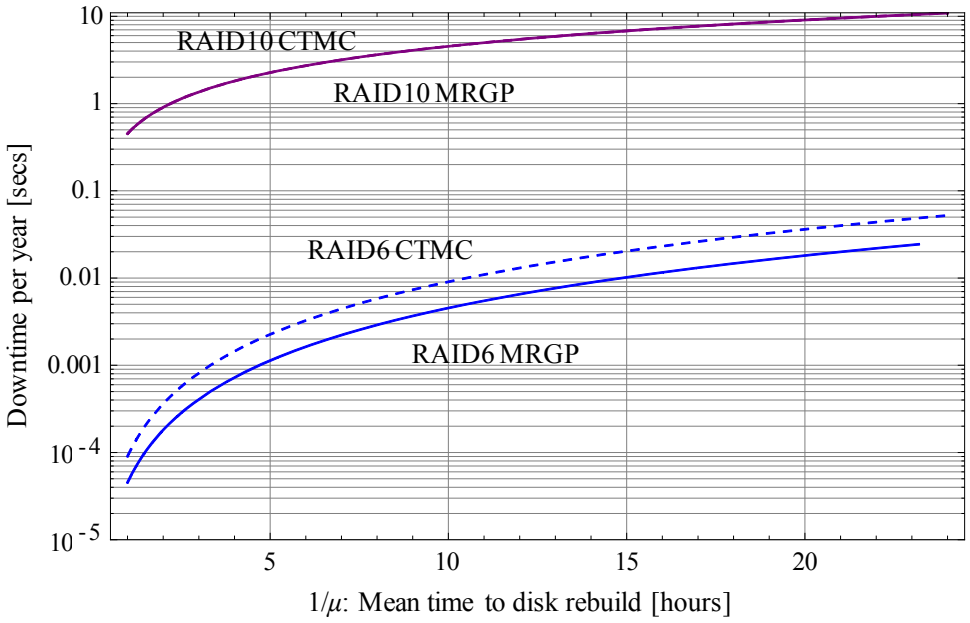


Figure 5.7 Comparison of downtime computed by CMTC and MRGP models while varying disk rebuild time [99]

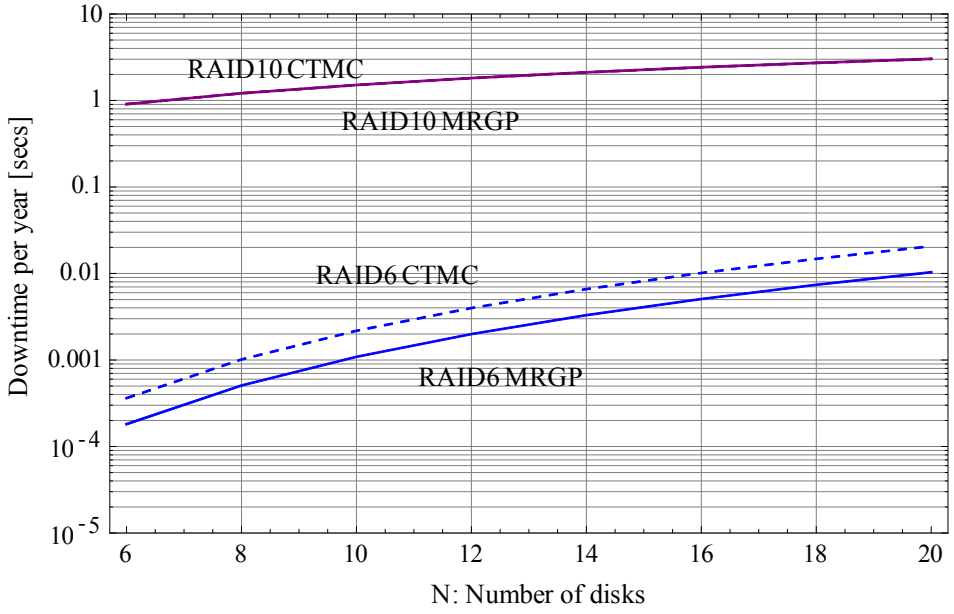


Figure 5.8 Comparison of downtime computed by CMTC and MRGP models, varying the number of disks [99]

Finally, we conduct the sensitivity analysis of the number of disks by setting $\lambda = 10^{-6}$ and $\mu = 0.5$. In general, as the number of disks increases, the storage reliability becomes worse because of the increased failure rate. Figure 5.8 shows the results of the estimated downtime for RAID6 and RAID10. As the number of disks constituting RAID6 increases, the difference between the computed downtimes becomes larger. However, the difference is very small (<0.01 second) even when the number of disks is 20. For the case of RAID10, the difference is less than 0.01 second in the whole range of the graph. From the above observations, the CTMC model tends to overestimate the downtime due to its assumption of memoryless property for the disk rebuild time. However, the difference is generally negligible in practical ranges of disk failure rates and disk rebuild times.

5.2.1.5.3. Performability comparison

The second part of our experiments focuses on the performability of RAID storage systems. In addition to data availability, the performance degradation caused by disk failures should be considered in the design of storage configurations. To quantify the performance degradation, first we conduct disk benchmarks on our experimental RAID storage systems.

Our test system comprises two physical servers equipped with the same hardware components including hardware RAID controller. One server is configured as a RAID6 system with six disks and another server is configured as a RAID10 with the same number of disks. All the disks used in the test system have the same specification produced by the same vendor. The disk benchmark is performed with fio (http://freecode.com/projects/fio), a tool for disk benchmarking. In our experiments, fio creates four jobs that continuously issue I/O requests of 1MB block with specified access pattern; either sequential read/write or random read/write. In order to apply the benchmark results to the degraded RAID storage system, we emulate disk failures by manually ejecting disks from the servers.

Figure 5.9 summarizes the benchmark results that show the average bandwidth measured by fio with different access types (i.e., sequential read/write or random read/write) for each degradation level of RAID6 and RAID10 storage systems.

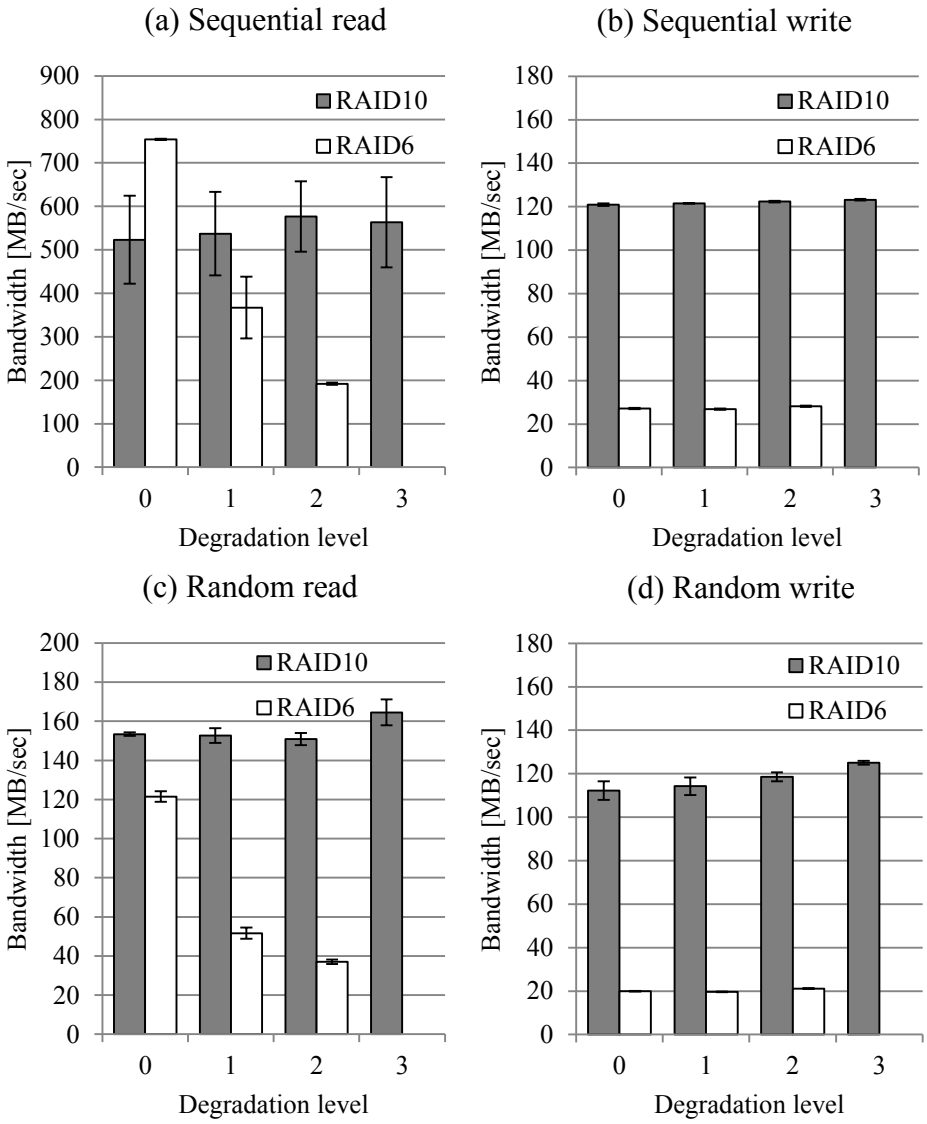


Figure 5.9 Benchmark results using fio; (a) bandwidth for sequential read, (b) for random read, (c) sequential write and (d) random write [99]

The degradation level corresponds to the number of failed disks in the storage system. Note that RAID6 tolerates any two disk failures, while RAID10 might tolerate up to triple disk failures. An interesting observation is that the read performance of RAID6 is decreased considerably after disk failures (in both sequential and random accesses). In particular, sequential read performance of RAID6 in state 0 is 44% higher than that of RAID10, but the

performance advantage is overtaken by RAID10 after one disk failure. The considerable performance degradation is caused by the decrease in striping level due to disk failures in RAID6; meanwhile the striping level of RAID10 is not reduced as long as RAID survives. On the other hand, the write performance of RAID6 is not decreased significantly by disk failures because the level of striping has relatively small impact on the write overhead including parity generation.

The benchmark results are then used for reward assignment in our MRRM models. Specifically, we assign the read access throughput values in MB per second at different degradation stages to the corresponding MRGP states in RAID6 and RAID10 models. We compute the performability of read accesses only, because the write performance is not much influenced by disk failures as presented in the benchmark results. We set the mean disk rebuild time to two hours and 24 hours and vary the mean time to disk failure from a thousand hours to a million hours. Figure 5.10 and Figure 5.11 show the computed performability for sequential read access and random read access, respectively.

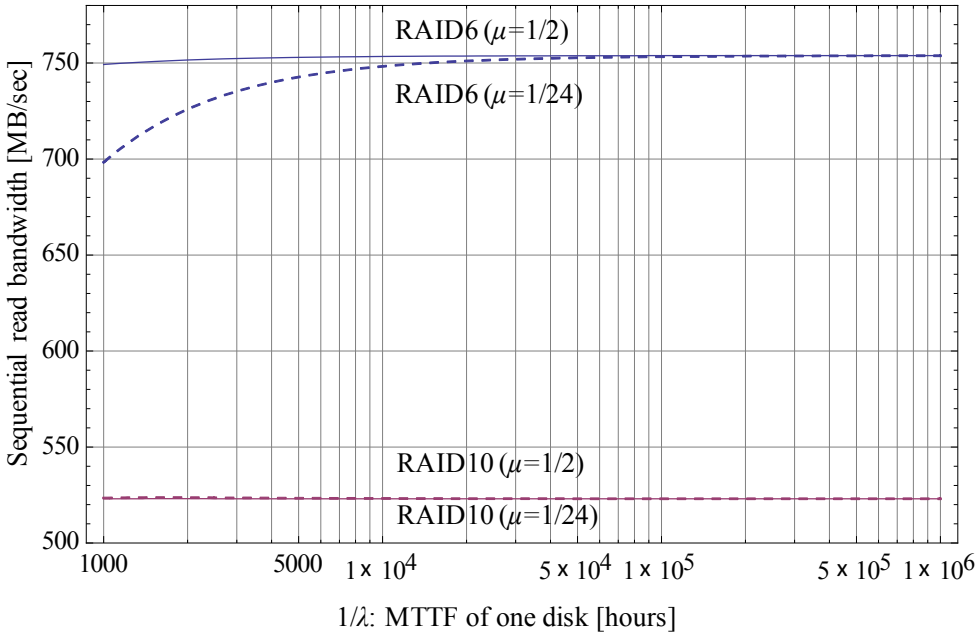


Figure 5.10 Performability of sequential read access in RAID6 and RAID10 systems [99]

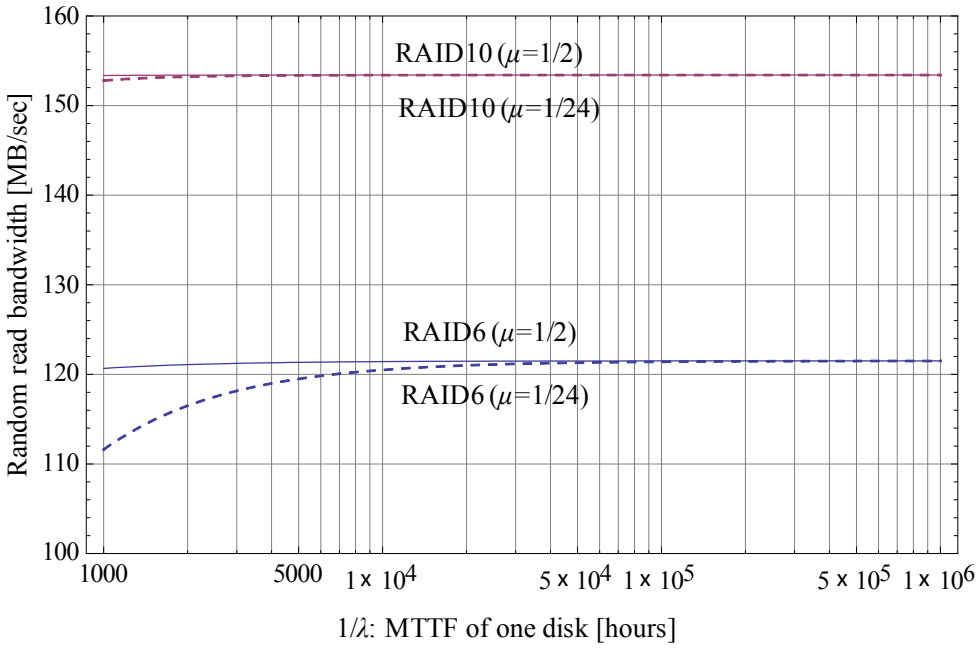


Figure 5.11 Performability of random read access in RAID6 and RAID10 systems [99]

Despite the significant performance degradation after disk failure in RAID6, the sequential read performance of RAID6 still has advantage over RAID10. On the other hand, the random read performance of RAID6 is apparently worse than that of RAID10.

Next we compare the results with the performability predicted by CTMC models. Using the expressions presented in [109], we compute the difference $\Delta P_i (i \in \{\text{RAID6}, \text{RAID10}\})$ between the predicted performability by MRGP and the value of the performability computed by CTMC. Figure 5.12 and Figure 5.13 show the performability differences of sequential read access and random read access, respectively. When we look at Figure 5.13, there is a change point around λ=1120 in ΔP_{RAID6} by $\mu = 1/24$. The difference ΔP_{RAID6} increases when $1/\lambda < 1119$, while it starts decreasing when $1/\lambda > 1120$.

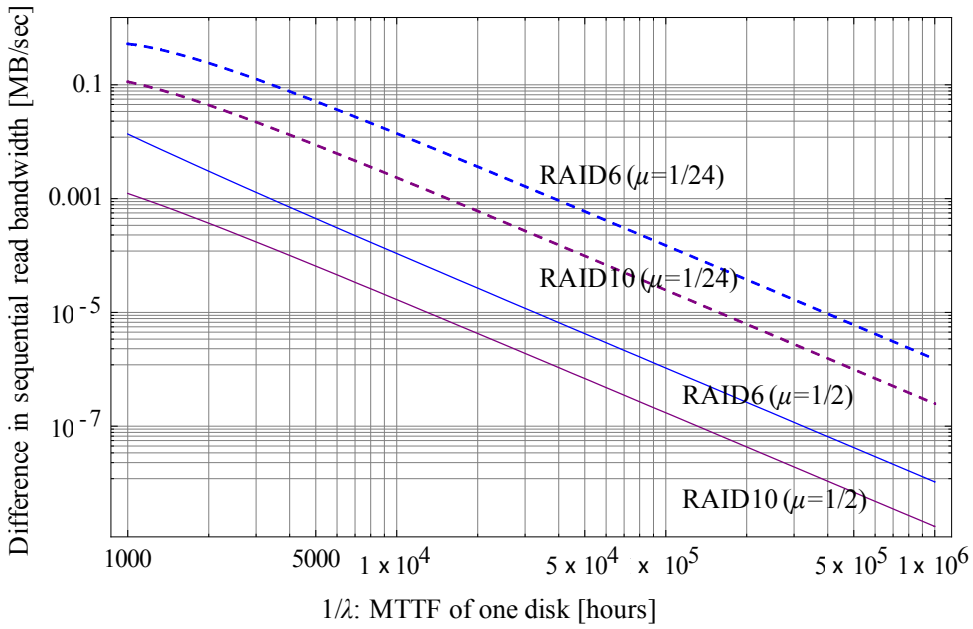


Figure 5.12 Difference in performability prediction of sequential read access between MRGP and CTMC [99]

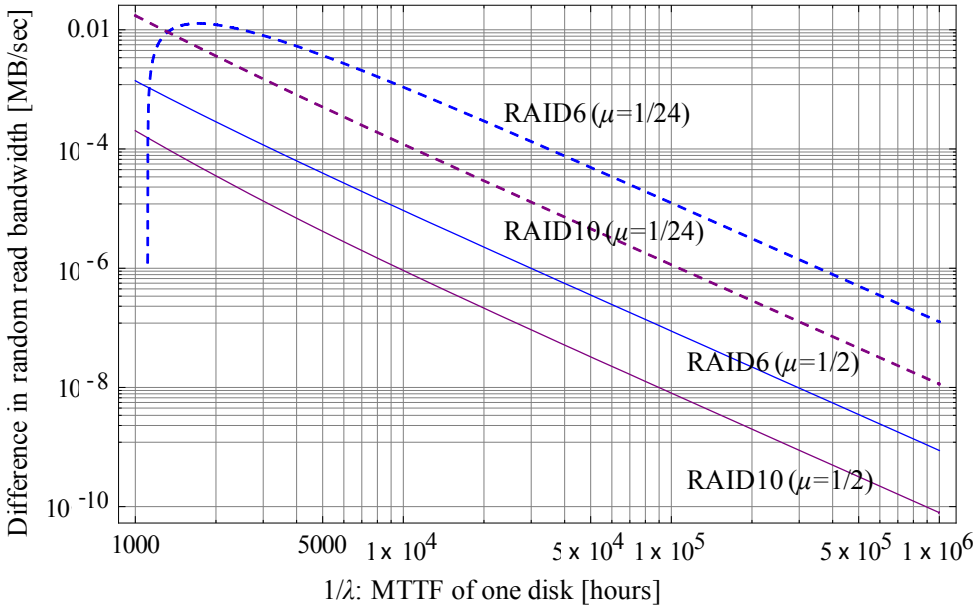


Figure 5.13 Difference in performability prediction of random read access between MRGP and CTMC [99]

In general, the difference becomes small as the mean time to disk failure increases. When we assume the mean time to disk failure as 10^6 hours for instance, the performability differences among the RAID models (CTMC vs. MRGP) are less than 10^{-5} MB/sec in sequential read access and are less than 10^{-7} MB/sec in random read access, regardless of the rebuild rate and RAID architecture (i.e., RAID6 or RAID10). In both the sequential and random access cases, the difference observed among the RAID10 MRGP and CTMC is generally smaller than that among the RAID6 MRGP and CTMC. Also, we can see that the smaller rebuild rate ($\mu = 1/24$) generally causes the larger difference between the results of CTMC and MRGP especially when the mean time to disk failure is larger than 2000 hours.

The relative difference, computed from the difference ΔP_i divided by the performance in the robust state (r_0), is less than 0.01 in all the range of the mean time to disk failure (not presented in the graphs). In contrast to the data availability prediction studied in the previous section, the approximation error in performability prediction from CTMC may not be negligible especially when the performance drastically changes in degraded states. Our MRGP model provides more accurate prediction of performability in such cases.

5.2.1.5.4. Sensitivity to rebuild time distribution

In the above comparative study, we used deterministic rebuild time in MRGP models. Although we do not see much variance in disk rebuild time in our experimental system, the rebuild time may vary due to workload change or other external factors. To look into the impact of variance of rebuild time, we conduct another sensitivity analysis using gamma distribution for rebuild time with same mean ($1/\mu$=2). The probability density function of gamma distribution with mean value ($1/\mu$) is defined by

$$g(x) = (\beta\mu)^{\beta} x^{\beta-1} \frac{e^{-\beta\mu x}}{\Gamma(\beta)}, \quad \text{where } \Gamma(\beta) = \int_0^{\infty} x^{\beta-1} e^{-x}\, dx,$$

β (>0) is the shape parameter of the distribution and if β is an integer value the distribution represents an Erlang distribution. Note that the exponential distribution is the special case of the gamma distribution with $\beta = 1$.

We conduct a sensitivity analysis on RAID6 MRGP model by using the different parameter values $\beta \in [0.1, 0.5, 1, 2, 10]$. Figure 5.14 shows the mean time to storage failure by varying disk failure rate.

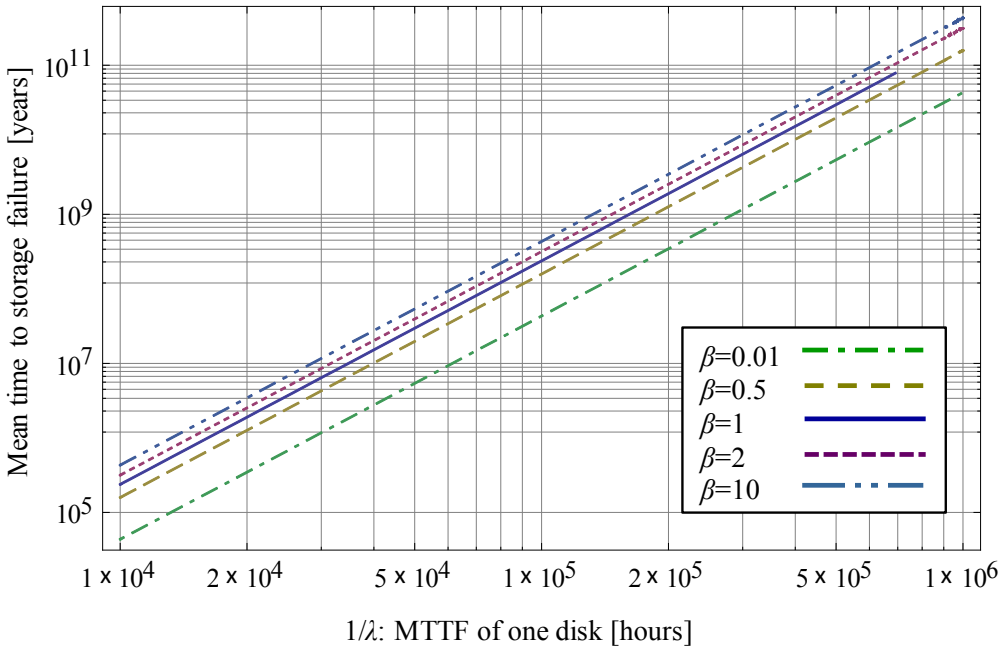


Figure 5.14 Sensitivity of rebuild time distribution on mean time to storage failure (in years) [99]

We observe that the mean time to storage failure becomes longer than the CTMC case (β =1) if β>1, while it becomes smaller if β<1. Since the second moment of the above gamma distribution is $1/(\beta\mu^2)$, the smaller value of β means larger variance that leads to lower mean time to storage failure, which agrees with the findings from [106]. Similarly, Figure 5.15 shows the results of the sensitivity analysis in terms of storage downtime. The downtime becomes larger than the CTMC case (β=1) if β<1, while it becomes smaller if β>1. Considering that the rebuild operation typically requires fixed amount of time at the minimum, the shape parameter β in reality tends to be large. In the extreme case, the distribution is unit step function (i.e., deterministic), as we assumed in the previous sections. In conclusion, although there is some impacts of rebuild time distributions on the reliability

and availability, the difference is generally negligible in the practical range of the parameter values.

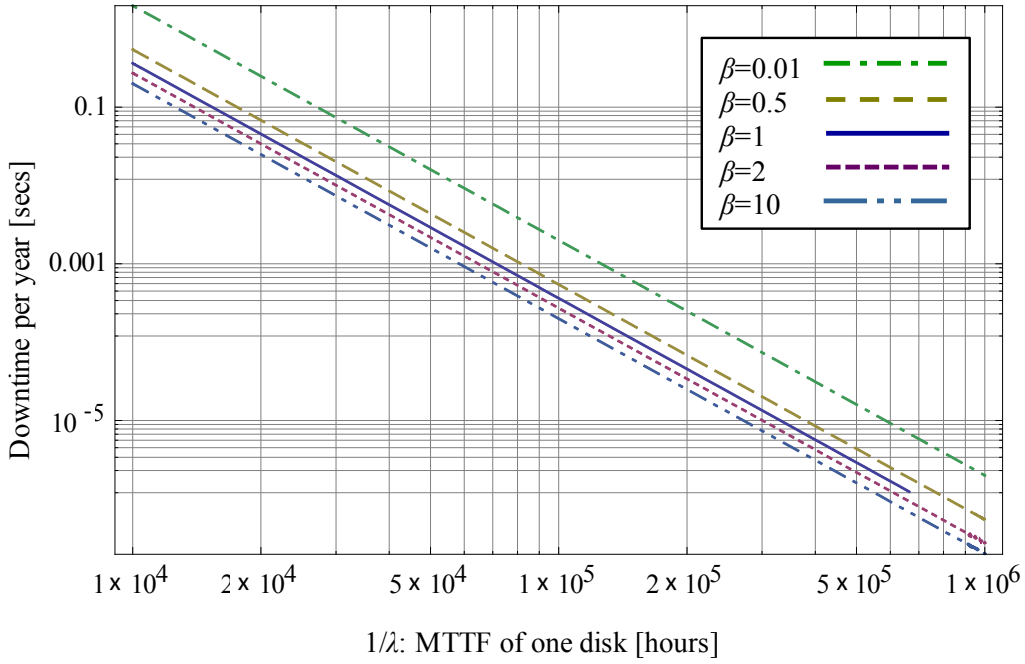


Figure 5.15 Sensitivity of rebuild time distribution on storage downtime [99]

5.2.1.6. Summary

In this study, we presented MRGP models for RAID storage systems. MRGP can express the time-dependent rates behavior of disk rebuild and storage reconstruction times and hence the memory-less assumption of the conventional RAID reliability models is relaxed. Nevertheless, our numerical study shows that the difference between the computed downtime by CTMC and that by MRGP is insignificant when we assume disk rebuild time is a few hours and the mean time to disk failure is in the rage of 10^4-10^6 hours. This implies that the memory-less assumption for disk rebuild time as in the CTMC models for RAID storage system is acceptable for practical use. On top of the MRGP models, we also presented the performability comparison between RAID10 and RAID6. Disk benchmark results are used to assign the reward rates for the MRGP models and the sensitivity to the mean time to disk failure is analyzed. Although RAID10 generally achieves better performance than RAID6,

we show the advantage of RAID6 in terms of performability in sequential read access. Compared with CTMC models, our MRGP model provide more accurate prediction of performability.

5.3. Data backup scheduling

5.3.1. A Markov decision process approach

This section describes the MDP-based approach to derive the optimal data backup schedule for multiple data sources in a data center. The preliminary work has been done in collaboration with Dr. Ruofan Xia and Prof. Kishor Trivedi in Duke University, and was presented in the workshop of DSN2014 and PRDC2015 [125][126]. In this thesis, MDP formulations are slightly revised so as to reduce the complexity while focusing on the backup schedule optimization.

Data loss caused by system failures or erroneous operations has significant impact on the operations of IT systems including business analytics used in industries these days. It is a critical issue for business owners to protect data efficiently and effectively. In a system with multiple data sets which have individual data protection requirements, backup planning plays an important role for maintaining the desired level of data protection while minimizing the impact on system operation. In this study, we investigate the use of Markov Decision Process (MDP) to guide the planning of data backup operations and propose a framework to automatically generate an MDP instance from system specifications and data protection requirements.

5.3.1.1. Introduction

Today businesses are increasingly relying on data analytics to provide insight into their operation and guide decision-making processes. Due to the increase in the data-intensive applications, maintaining their data properly in a data center becomes an important task for system management. Unfortunately, data can be lost due to various reasons [127]. Files may be corrupted by unintentional modification or malicious behaviors. Data may become inaccessible due to failures of storage system components. Natural disasters may destroy the

physical infrastructure which stores important data. Impact from such losses ranges from reduced productivity and capability to carry out normal operation, to significant financial losses, damaged business reputation and customer confidence [128]. Thus, system administrators or management tools need adequate protection for users' important data against such eventualities.

While recent distributed systems employ replication of the user data across geographically distributed sites, data backup are still necessary for users requiring data archive. Moreover, since data replication itself does not tolerate to data loss caused by operational errors or malicious activities, it is important to take backup of the data in a periodic manner. The frequency of data backup could depend on the users' business requirements to specify the acceptable damage due to data loss. Two frequently used requirement terms are Recovery Point Objective (RPO) and Recovery Time Objective (RTO) [127]. For a data source, RPO defines a time point in system history so that a recovery procedure must be able to recover the system status at that epoch. On the other hand, RTO specifies the maximum length of the recovery period that is acceptable. These requirements deal with the data loss and downtime aspects of data source failures respectively, and serve as part of the design and operational guidelines for the data sources. Backup scheduling plays an important role in meeting such requirements under administrative constraints, but it is typically a non-trivial task. The reason for this is two-fold. First, data backups typically have associated cost in terms of downtime and/or performance overhead. Backup execution may affect normal system operation, by suspending data access (for instance to maintain data consistency) or contending with production workload execution. Second, data backup must take into consideration various other factors such as data priorities and the available system resources for backup, which may be limited.

In this study, we investigate the application of Markov decision process (MDP) [25] to data backup scheduling. We present a framework that translates a system specification, consisting of different data sets and their required levels of protection, along with the amount of resource available for backup execution, into an MDP instance. The framework takes into

consideration the failure/recovery behaviors of system components and provides optimal backup operation scheduling so as to minimize downtime.

The organization of the following subsections is as follows. In Section 5.3.1.2, we briefly revisit the introduction of MDP employed in this study. Section 5.3.1.3 describes the system with multiple data sets to be backed up to protect data from unexpected data loss. The requirements for data protection as well as resource constraints to be satisfied are also discussed. Section 5.3.1.4 presents the framework to translate the system scenario into an MDP instance. Section 5.3.1.5 discusses related work. Finally, we provide a conclusion.

5.3.1.2. Markov decision process

The definition of Markov Decision Process is described in Section 3.3.1. As its name suggests, MDP possesses the Markov property in the sense that the system evolution beyond a decision point depends only on the system state and the action chosen at that point. The theory of MDP indicates that it is sufficient to locate a stationary policy to achieve optimality [25], meaning that there is no need to consider the past history when making a decision about which action to perform in a given state. Thus, the goal of solving an MDP is to generate a mapping from the states to their actions, i.e., a stationary policy, so that in a given state an action is known to be the optimal choice to control system evolution from that point. Once a stationary policy is obtained, the transitions (and their probabilities) from each state become fixed, and the MDP becomes a DTMC. The optimality in an MDP is in the sense of maximizing or minimizing some cost criteria during the evolution of the system. There are several common choices: the expected total cost, the expected discounted cost, and the expected time-averaged cost. The expected total cost criterion is applicable to finite-horizon MDPs where the number of decision points is finite. In contrast, the expected discounted cost and expected time-averaged cost criteria can be applied to infinite-horizon MDPs, where the number of decision points is infinite (but typically countable). In this study, we use the expected discounted cost as the objective function.

5.3.1.3. System configuration

In this study, we consider a storage system maintaining multiple *data sets* that are collections of files managed by middleware such as DBMS or distributed cloud data store. In normal operation, the data sets accept change to the data resulting from user operations on the data. The system is expected to store such data update securely. However, data sets may experience failures caused by software bugs in the program managing the data or failure of hardware components hosting data, which can lead to loss of the accumulated user update. In addition, failures of data sets and subsequent recovery incurs the downtime for data access, resulting in lower data availability. To protect data from such failures, data backup is taken place for each data set where the copy of the data sets is stored in a backup storage. The copied data in the backup storage can be used to restore the data when encountering failures of the data set in the main storage system. During the data backup and recovery operation, the users cannot access the data set.

Data backup operations are performed in the maintenance period of the system where typically workloads to data sets are not so heavy. At the beginning of the maintenance period, for each data set, the system may decide to start a backup operation or skip this backup opportunity and continue the normal operation. We consider this time instant as a decision point of data backup. There are two types of backups that each data set can choose at a decision point; full backup or a partial backup. For a full backup, all the data in the main storage is copied to the backup storage. On the other hand, a partial backup only copies the updated data from the latest backup data. Since the amount of data to be backed up is different, a full backup takes longer time than the time taken by a partial backup. When considering the recovery from failures, however, data restoration from the full backup data is faster because in case of recovery from the latest partial backup it requires to restore the latest full backup data and then apply all the partial updates to recover the latest data set.

Execution of a backup requires some system resource such as network bandwidth to transfer the processed data, and there may not be enough resource to accommodate concurrent backup execution for all data sets. This fact may require some data sets to backup more/less frequently. In this study, we consider a single type of shared resource, the backup

network bandwidth, but other shared resources can also be easily incorporated by adding similar constraints in the formulation.

At each decision point, the system needs to be determined which data sets should be taken backup in accordance with the requirements for data protection under the limited amount of shared resources. The requirements for data protection are specified by RTO and RPO. In our context, the RPO defines the maximum number of backups that a data set may consecutively skip, since if a data loss occurs the system cannot recover the status of data at a skipped backup point. As for RTO, it defines how many partial backups a data set may accumulate before a full backup should take place, as each additional partial backup to process during recovery increases the overall recovery time. The backup schedule should satisfy all the RPO and RTO requirements of individual data sets.

Since data sets may fail, the backup planning should take into account their failure and recovery behavior. We model the state transition of each data set by a semi-Markov process as shown in Figure 5.16.

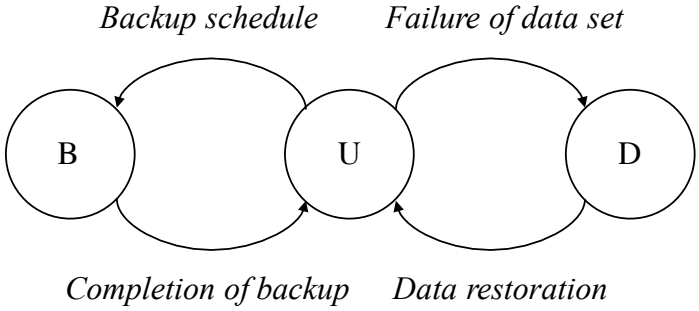


Figure 5.16 Availability model for a data set

The data set is accessible and in the normal operation at the up state *U*. When the data set encounters a failure, the state is changed to the down state *D*. The data set remains in the *D* state until the latest data is restored from the latest backup. Once a backup operation starts, the data set becomes inaccessible that is represented by the backup state *B*. We assume that the failure time of the data set is exponentially distributed with rate λ, while data restoration and backup completion are completed in the deterministic time intervals that are determined

by the type of backup and the amount of data to be restored. The transition from state U to state B depends on the given backup schedule which is discussed in the next section.

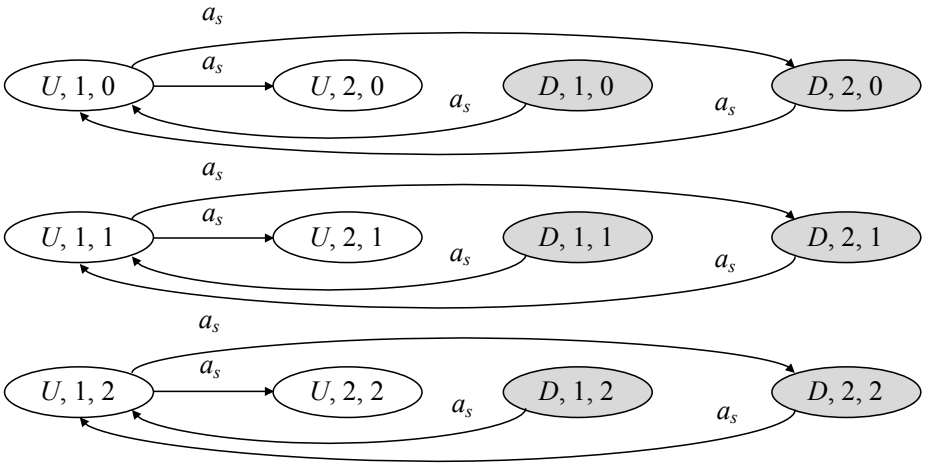
5.3.1.4. Formulation of backup scheduling by MDP

5.3.1.4.1. Status of each data set

The state space of the MDP is constructed from the status of the data sets. Denote the status of data set i as (A_i, X_i, Y_i) where A_i represents the failure status of data set i, X_i is the number of decision intervals since the last backup of data set i, and Y_i is the number of partial backups taken since the last full backup. The variable A_i is either U or D according to the availability model in Figure 5.16. The second variable is the number of successive backup points that the data set has skipped. The maximum values of X_i and Y_i are specified by the values of RPO_i and RTO_i respectively, which are the RPO and RTO requirements of data set i.

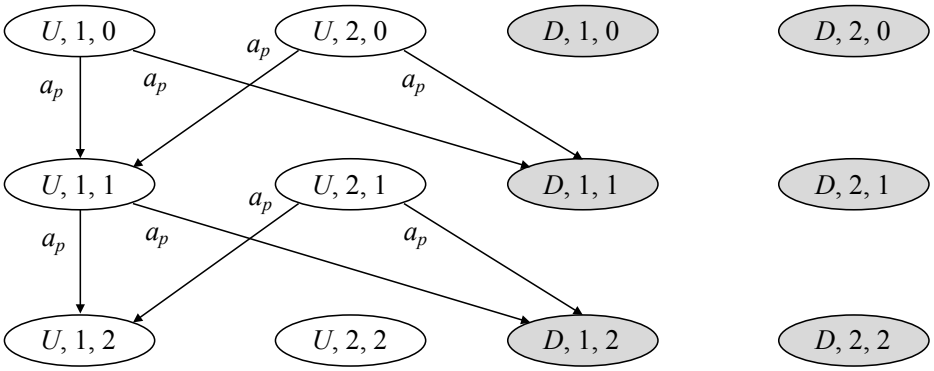
The action of the MDP is constructed from the backup choices of each data set at a decision point. There are three actions each data set can take; skip this backup and continue normal operation, take a partial backup, or take a full backup. We denote these actions a_s, a_p and a_f, respectively. The RPO and RTO requirements dictate whether each action is possible under the current state for that data set. For example, if a data set has reached its maximum allowed number of skipped backups then it must perform either partial backup or full backup, and if it has also reached its maximum number of partial backups then a full backup is the only choice.

A small MDP example, composed of one data set with RPO=2 and RTO=2, is shown in Figure 5.17, Figure 5.18, and Figure 5.19 where the status transition enabled by a_s, a_p and a_f, respectively, are presented. Notice that while in general three actions are available in a status, in some statuses the set of actions becomes restricted. For example, in $(U, 2, 0)$ a backup must be taken, while in $(U, 2, 2)$ only full backup is possible. Allowable status transitions are detailed with the transition probabilities in the following section.



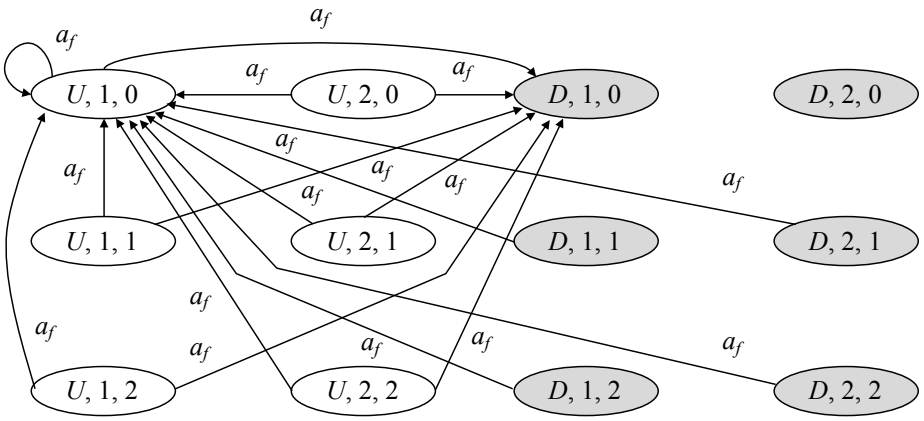
Status (A, X, Y): A: State of the data set (U or D)
X: Decision interval counter from the last backup
Y: Number of partial backups performed

Figure 5.17 Status transitions enabled by skip action for a data set with RPO=2 and RTO=2



Status (A, X, Y): A: State of the data set (U or D)
X: Decision interval counter from the last backup
Y: Number of partial backups performed

Figure 5.18 Status transitions by partial backup for a data set with RPO=2 and RTO=2



Status (A, X, Y): A: State of the data set (U or D)
X: Decision interval counter from the last backup
Y: Number of partial backups performed

Figure 5.19 Status transitions by full backup for a data set with RPO=2 and RTO=2

5.3.1.4.2. Transition probabilities and costs

The transition probabilities and costs of MDP actions are also constructed from those of each data set. For a data set, the cost is the downtime that this data set can expect to incur during its operation. Denote T as the time interval between the consecutive backup decision points. Let d_p and d_f be the downtimes due to partial and full backup, respectively. When a data set is failed, the data needs to be recovered from backup data. We assume the time for data set restoration is given by $d_r(y) = \delta_f + y \cdot \delta_p$, where y is the number of partial backups taken from the last full backup, δ_f and δ_p are the time time to restore full backup data and a single partial backup data, respectively. The transition probability and the cost for each status of a data set can be defined as follows.

1) When the current data set is available, the current status can be specified as (U, x, y), where x is any positive integer and y is any nonnegative integer according to the backup state.

a. If the skip action a_s is chosen, the possible status in the next decision point is either (U, x+1, y) or (D, x+1, y), which correspond, respectively, to the case of the data set being down or up at the next decision point. The status (U, x+1, y) can be achieved when there is no failure between the consecutive decision points or the data set is failed but is recovered before the next decision point. The transition probability from the status (U, x, y) to the status (U, x+1, y) with the choice of skip action can be given by

$$\begin{aligned}&\Pr[a_s; (U,x,y),(U,x+1,y)|w=0] + \Pr[a_s; (U,x,y),(U,x+1,y)|w=1]\\&= e^{-\lambda T} + \left[1 - e^{-\lambda(T-d_r(y))}\right]\\&= 1 - e^{\lambda T d_r(y)},\end{aligned}$$

where $w \in \{0,1\}$ represent the number of failures during this period. On the other hand, when the data set fails and it is not recovered at the next decision point, the status is changed to (D, x+1, y). The corresponding transition probability is

$$\Pr[a_s; (U,x,y),(D,x+1,y)|w=1] = e^{-\lambda(T-d_r(y))}\left[1 - e^{-\lambda d_r(y)}\right].$$

Since the skip action does not incur the downtime due to backup operations, the associated cost is simply determined by whether it experiences a failure or not. The costs are given by

$$\begin{aligned}C[a_s; (U,x,y),(U,x+1,y)|w=0] &= 0\\C[a_s; (U,x,y),(U,x+1,y)|w=1] &= d_r(y)\\C[a_s; (U,x,y),(D,x+1,y)|w=1] &= d_r(y).\end{aligned}$$

b. If the full backup action a_f is chosen, the possible status in the next decision point is either (U, 1, 0) or (D, 1, 0) depending on whether the data set is down at the next decision point. The status (U, 1, 0) can be reached when the data set is not failed during the backup period or recovered before the next decision point. Since we assume that the data set is not failed during the backup operation, the failure can occur only after the completion of full backup. Therefore, the transition probability is given by

$$\Pr\left[a_f; (U,x,y), (U,1,0) | w = 0\right] + \Pr\left[a_f; (U,x,y), (U,1,0) | w = 1\right]$$
$$= e^{-\lambda(T-d_f)} + \left[1 - e^{-\lambda\left(T-d_f-d_r(y)\right)}\right]$$
$$= 1 - e^{\lambda T d_r(y)}.$$

On the other hand, the status (D, 1, 0) is reached only when the data set fails after the full backup and is still being in down condition at the next decision point. We assume that the data set is not failed during the backup operation. Thus, the transition probability to (D, 1, 0) is given by

$$\Pr\left[a_f; (U,x,y), (D,1,0) | w = 1\right] = e^{-\lambda\left(T-d_f-d_r(y)\right)}\left[1 - e^{-\lambda d_r(y)}\right].$$

The full backup operation incurs the downtime d_f, thus the accosted cost is determined by

$$C\left[a_f; (U,x,y), (U,1,0) | w = 0\right] = d_f$$
$$C\left[a_f; (U,x,y), (U,1,0) | w = 1\right] = d_f + d_r(y)$$
$$C\left[a_f; (U,x,y), (D,1,0) | w = 1\right] = d_f + d_r(y).$$

c. Similar to the above case, if the partial backup action a_p is chosen, the possible status in the next decision point is either (U, 1, y+1) or (D, 1, y+1) depending on whether the data set is down at the next decision point. The downtime caused by backup is d_p instead of d_f. The transition probabilities are

$$\Pr\left[a_p; (U,x,y), (U,1,y+1) | w = 0\right] + \Pr\left[a_p; (U,x,y), (U,1,y+1) | w = 1\right]$$
$$= e^{-\lambda(T-d_p)} + \left[1 - e^{-\lambda\left(T-d_p-d_r(y)\right)}\right]$$
$$= 1 - e^{\lambda T d_r(y)},$$

and

$$\Pr\left[a_p; (U,x,y), (D,1,y+1) | w = 1\right] = e^{-\lambda\left(T-d_p-d_r(y)\right)}\left[1 - e^{-\lambda d_r(y)}\right].$$

The associated costs are given by

$$C\left[a_p; (U,x,y), (U,1,y+1) | w = 0\right] = d_p$$
$$C\left[a_p; (U,x,y), (U,1,y+1) | w = 1\right] = d_p + d_r(y)$$

$$C\left[a_p; (U, x, y), (D, 1, y+1) | w = 1\right] = d_p + d_r(y).$$

2) When the current data set is not available, the status is represented as (D, x, y). To simplify the problem, we neglect the possibility that the data set encounters another failure in the next decision period assuming that such a probability is marginal. A full backup can be schedule at this decision point after the recovery from backup data if $y>0$. The full back can reset the value of y. On the other hand, a partial backup cannot be scheduled at this moment because there are no updates to be backed up after the recovery.

   a. If the skip action a_s is chosen, the system transits to the status (U, 1, y) with probability one. Since there is no additional downtime, the associated cost is zero. Note that the recovery time from the down state to up state is included in the cost of previous decision point.

   b. If the full backup action a_f is chosen, a full back operation is performed after the completion of recovery and the system transits to (U, 1, 0) with probability one. The associate cost is the downtime due to a full back d_f.

Table 5.3 summarizes the transition probabilities and associated costs defined for the current statuses and the selected actions. Transition probabilities and costs are not defined for the actions which are restricted under specific conditions. For skip action a_s, it is prohibited when x is equal to RPO, since either full or partial backup is necessary to meet the RPO requirement for the data set. For full backup action a_f, when the current status of the data set is down and the value of y is equal to 0, it does not necessary to perform a full backup because there are no updates after the recovery. For partial backup action a_p, if the data set is under the recovery process, the latest data must be restored at the completion of recovery and hence it does not necessary to take another partial backup. In addition, partial backup cannot be chosen when the value of y reached to RTO, since a full backup is necessary to meet the RTO requirement for this data set.

Table 5.3 Transition probabilities and costs associated with MDP actions

Action	Current status		Fail	Next status	Transition probability	Cost
a_s	(U, x, y),	x<RPO	0	$(U, x+1, y)$	$e^{-\lambda T}$	0
			1	$(U, x+1, y)$	$1 - e^{-\lambda(T-d_r(y))}$	$d_r(y)$
			1	$(D, x+1, y)$	$e^{-\lambda(T-d_r(y))}\left[1 - e^{-\lambda d_r(y)}\right]$	$d_r(y)$
	(U, x, y),	x=RPO	-	-	-	-
	(D, x, y)		0	$(U, 1, y)$	1	0
a_f	(U, x, y)		0	$(U, 1, 0)$	$e^{-\lambda(T-d_f)}$	d_f
			1	$(U, 1, 0)$	$1 - e^{-\lambda\left(T-d_f-d_r(y)\right)}$	$d_f + d_r(y)$
			1	$(D, 1, 0)$	$e^{-\lambda\left(T-d_f-d_r(y)\right)}\left[1 - e^{-\lambda d_r(y)}\right]$	$d_f + d_r(y)$
	(D, x, y),	y>0	0	$(U, 1, 0)$	1	d_f
		y=0	-	-	-	-
a_p	(U, x, y),	y<RTO	0	$(U, 1, y+1)$	$e^{-\lambda(T-d_p)}$	d_p
			1	$(U, 1, y+1)$	$1 - e^{-\lambda\left(T-d_p-d_r(y)\right)}$	$d_p + d_r(y)$
			1	$(D, 1, y+1)$	$e^{-\lambda\left(T-d_p-d_r(y)\right)}\left[1 - e^{-\lambda d_r(y)}\right]$	$d_p + d_r(y)$
		y=RTO	-	-	-	-
	(D, x, y)	-	-	-	-	-

As introduced in the system configuration, the system consists of multiple data sets each of which has own RTO and RPO requirements. Thus, MDP state is composed of the statuses of all the data sets, the possible actions in a given MDP status are also constructed as the combination of individual data set actions. For example, if a system consists of two data sets, $ds1$ and $ds2$, which take action a_1 and a_2 in state s_1 and s_2, respectively, then the corresponding MDP action in state (s_1, s_2) would be (a_1, a_2). One additional constraint put on the MDP actions is that the number of data sets executing backup in one decision point cannot exceed the capacity of system backup resource. To simplify discussion, we assume a backup

operation consumes one unit of resource regardless of the set executing it. This can be easily relaxed by allowing some backup executions to consume more time.

The transition probabilities and costs are also defined on the MDP with multiple data sets. While the transition probabilities are obtained by multiplying the probabilities of entering corresponding new statuses from all data sets, the cost is computed by the average of those from all the data sets.

5.3.1.4.3. MDP construction procedure

The state-space and actions, along with the costs and transition probabilities, complete the specification of the MDP instance. Based on these elements, an MDP can be constructed following the procedure.

1) Construct the MDP state-space by taking a Cartesian product of statuses of all the data sets.
2) Generate a reachability graph from the MDP states, where a state can reach another state if there is a valid action (i.e., one that does not exceed resource constraint) that will cause the system to transition from the former to the latter.
3) Prune the state space by a depth-first search through the reachability graph. Any state that cannot reach itself is a state on a path leading to RPO and/or RTO violation. All such states are removed from the MDP state-space.
4) Compute the transition probabilities and the associated costs for all the possible transitions in the MDP.

The constructed MDP instance is used to derive the backup schedule in the form of optimal policy which assign the optimal action for each MDP state so that average downtime of all the data sets is minimized.

5.3.1.5. Related work

Due to the importance of data protection, there has been much work devoted to this topic. Some research work focuses on design and implementation of new techniques to improve

effectiveness of data backup and other protection techniques, for example [129] and [130] focus on the technique of data de-duplication, while [131] and [132] investigate online backups. Some other works provide overview on general issues and techniques in specific contexts such as [133]. There are also works that focus on modeling and evaluation of data backup techniques and strategies, such as [134][135][136]. Our work here differs from these examples in that we focus on designing of optimal data backup schedule.

There are also works on effective design and/or management of data backup operations. Many of these works utilize some form of optimization. For example, [137] considers optimal data placement and level of replication, while [138] investigates the design of a storage solution for a specific context. By comparison, our work focuses on backup scheduling, with the application of MDP which allows a large set of practical scenarios to be modeled and optimized for.

5.3.1.6. Summary

In this section, we presented MDP approach to derive the optimum data backup schedule for given data recovery objectives. The presented framework allows the translation of several data- and system-related requirements into an MDP instance, so that the solution to the instance provides the optimal schedule that minimizes system downtime while satisfying the requirements.

In [125], some numerical investigation results were presented where the formulation of the transition probabilities and costs are slightly different from the content in this thesis. In a subsequent study [126], the scalability issue of the approach is addressed. In order to improve the scalability of MDP approach, the decomposition with approximation technique was introduced.

Chapter 6

Cloud resource management

This chapter focuses on the issue of resource management for cloud computing systems. In cloud computing, user applications are hosted on a common computing infrastructure consisting of clusters of servers and virtual machines. Resource management is one of the dependability issues with cloud computing systems, which can affect user-perceived performance as well as service unavailability. To avoid resource contention due to insufficient resources to meet user demands, effective resource management in the operation of cloud computing is essential. Significant effort has been made in developing virtual machine level resource management techniques because one of the major benefits of cloud computing is the capability of elastic resource allocation using virtual machine creation and migration, while host level resource management has not extensively been addressed. In practice, however, host level resource management is also an important issue when considering the long-range operational cost. Along with the growth of a cloud service, the number of users and their workloads can increase over time and may reach capacity limit where virtual machine level resource management cannot deal with increased demand. Unless additional server resources are supplied to the cloud system, user applications may encounter resource contention, which may lead to service unavailability. This chapter addresses this issue, in particular, additional server procurement decision for avoiding resource contention in a cloud computing system. As a specific instance of a cloud computing service, a mobile thin-client service provided in a private cloud system is considered. The procurement decision problem is formulated and a framework to guide procurement decisions are presented.

6.1. Cloud server procurement

6.1.1. Server procurement decision framework

This section presents a method to assist server procurement decision for mobile thin-client service hosting virtual machines (VMs) for mobile OS instances in a private cloud system. A part this section will be published as a book chapter of "Stochastic Operations Research in Business and Industry" in World Scientific, Singapore, in 2017 [140].

For an owner of a private cloud system for mobile thin-client service, a timely decision of server procurement is a key challenge to maintain the service performance while minimizing the cost of ownership. Since the procured server becomes a dead stock until it is actually used in the service, the procurement needs to be determined by forecasting future resource requests in consideration with the trend of VM demands, changes in VM workloads and the dynamics of VM resource allocation in the cloud system. To guide a better procurement decision, we propose a framework that combines the techniques for demand estimation of new VMs, workload estimation of hosted VMs, and repetitive simulation of VM replacement algorithm. The VM demand arrivals and workload changes are stochastically modeled, and the dynamics of resource allocation governed by VM replacement algorithm is simulated in the framework. We also conduct simulated experiments that show the proposed framework can reduce the total cost by up to 71% compared with a heuristic approach preparing a standby server all the time.

6.1.1.1. Introduction

Mobile thin-client service is a cloud-based service to provide computing resources for mobile OS instances used typically in companies who concern about the security of data [141]. Smartphones and tablets are widely accepted for business use, not only for communication tools but also for terminal devices for remote working. Remote users can access to the company's resources or services by smart devices from their homes or customer sites as long as the devices are connected to mobile networks. Enterprises allowing such a flexible use of smart devices, however, usually confront a security risk of lost or theft of the devices. If any

smart devices storing important confidential data are lost, it may lead to a serious consequence of information leak through the lost device. To avoid such a risk, mobile thin-client service provides a solution by executing mobile OS instances on virtual machines (VMs) running on a secure private cloud datacenter and allowing authorized access to the instances from mobile devices. Since the proprietary data is stored in the secure datacenter and is accessible via secure communication, the risk of information leak is significantly reduced even when the mobile devices are lost or theft.

Companies employing mobile thin-client service typically need to own a private cloud environment in a secure site. For the operation of the service, sufficient server resources for the private cloud are required for accommodating mobile OS instances. Insufficient sever resources cause the lower service level that might inhibit the productivity of business using mobile devices. On the other hand, unnecessary server resources impose the cost of ownership. Once a host server is procured for the system, it becomes an idle stock of resource, which is regarded as a wasted cost until the server is really in operation. Therefore, the number of servers needs to be adjusted according to the actual demands of the service. A timely decision of server procurement is a key challenge to reduce the resource cost of system owners.

This study extends the previous study [141] that presents a framework to assist the decision of timely server procurement by estimating VM demand arrivals, workload changes and VM resource reallocation. The previous study focuses on minimizing the server unused period (UNP) under the given service requirements. Avoiding service level violations is considered as a strict constraint that prior to the allowable cost of unused servers. In this paper, we reformulate the problem as an optimization problem to minimize the total cost including both of the factors of unused servers and service level violations. To deal with several uncertainties affecting cloud systems, stochastic models are introduced to capture the system dynamics.

In order to provide an effective solution to server procurement decision problem, the dynamics of VM resource allocation in a cloud must be taken into account. Based on a practice, we assume that VMs for mobile OS instances are reallocated among host servers on

the cloud at a certain time interval (e.g., everyday). The resource reallocation is handled by VM replacement algorithm packing VMs into the available capacities of host servers. Due to the increase in the VM instances and the changes in their workloads, the algorithm eventually might not find any solutions satisfying given constraints and then a new server is required. A difficulty in the decision of server procurement is partly caused by such a dynamic behavior of internal resource reallocation, which usually relies on heuristics and results in an ad-hoc placement. Depending on the algorithm or policy to VM reallocation, the time when a new server is required significantly changes. Such problem has not been addressed in the research of mobile cloud computing [139]. To address this issue, we exploit a simulation of VM replacement algorithm in the framework so as to predict the time to resource contention resulting in additional server requirement.

The effectiveness of the proposed framework is presented through the experiments in comparison with decision methods relying on heuristics. Our experimental results show that the proposed decision framework outperforms the heuristic approaches in terms of the total cost.

The rest of the section is organized as follows. Section 6.1.2.2 introduces a configuration of mobile thin-client service. Section 6.1.2.3 formulates the server procurement decision problem. Section 6.1.2.4 describes the system models including VM demand arrivals, workload changes and host server requests. Section 6.1.2.5 presents a procurement planning framework in which VM replacement algorithm is simulated based on the system models. Section 6.1.2.6 shows the simulated experimental results and Section 6.1.2.7 discusses the related work. Finally, Section 6.1.2.8 provides our conclusion.

6.1.1.2. Mobile thin-client service

Increasing concerns for security of mobile devices for enterprise use expand the demands to thin-client services for mobile devices like smartphones. Mobile thin-client service offers computing resources in a secure datacenter to mobile devices through encrypted network communication. Operating systems for smart devices are installed in VMs and their images and data are saved in the datacenter. A security policy prohibits the smart device users from

saving any confidential data in the local storage and hence the company can minimize the risk of information leak due to accidental lost of the mobile device. All the processes associated with company activities, such as mailer, scheduler and document viewers, are run in the datacenter environment instead of the mobile device. User inputs on the smart device are transferred to the corresponding VM instance in the datacenter and, on the other side, the only screen data is transferred to the mobile device (see Figure 6.1). The content of user inputs and screen data is protected by encrypted communication.

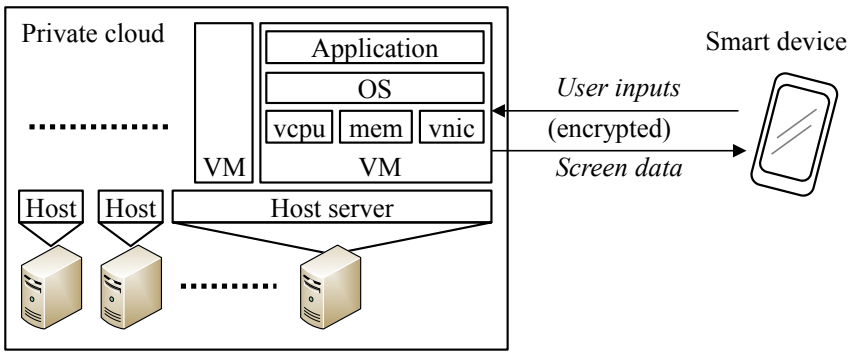


Figure 6.1 A service architecture of mobile thin-client service [141]

Applications and an operating system running on a virtual machine consume shared computing resources in the datacenter. Increased number of VMs might cause resource contention which results in performance degradation of user applications. User-perceived performance is affected not only by resource contention in the datacenter (e.g., CPU, memory and disk I/O) but also by end-to-end network bandwidth between the datacenter and the mobile device.

Following to the previous study, we assume that the bottleneck of the service performance exists in computing resources rather than network or storage I/Os in the datacenter. In order to keep a good performance of smartphone service, it is important to avoid CPU resource contention proactively by adding server resources for hosting VMs. Meanwhile unused server incurs unnecessary cost. To reduce the idling periods of server resources, a virtualized datacenter typically employs a heuristic approach to optimize VM placement such that all the workloads of the hosted VMs are packed in the given server resources.

Although VM placement optimization can help postpone the time to reach a resource contention, a new additional server is eventually required under the assumption the number of users gradually increases. An additional server should be prepared before the system actually encounters a resource contention. Thus, deciding timely server procurement is an important issue which is detailed in the next.

6.1.1.3. Server procurement decision

Server procurement needs to be scheduled in consideration with the tradeoff between the cost associated with the unused stock and the expected performance penalty due to the lack of sufficient resources. Premature decision of procurement causes "dead stock" where server resources remain unused for a long time. In the worst case, procured server resources are never used when the demands of VMs do not increase as expected. On the other side, delayed decision of server procurement results in resource depletion of server infrastructure and may violate the service level agreement (SLA) in terms of user-perceived quality such as an end-to-end response time. Therefore, the decision should be done in a timely manner with watching the trend of the demand increase. This decision problem can be regarded as a variation of classical inventory management problem with respect to the following two cost factors.

- Server unused period (UNP): It represents the time period between the time when a procured server is ready for use and the time when the server actually starts being used in the system. In order to reduce the unused resources and maximize the efficiency, the server unused period should be minimized.

- CPU overload: CPU overload event occurs when the average CPU utilization of a server in a time window exceeds a predefined threshold value. CPU overload is caused by the lack of computation resources and it degrades the quality of the services hosted on the server. Any additional server should be ready before encountering CPU overload.

Both of UNP and CPU overloads incur some operational costs that are affected by the time of server procurement decision. Let $T_{\text{unp}}(t)$ and $N_{\text{over}}(t)$ be the UNP and the number of CPU overloads observed in the system for time interval *t* from when the last server is added to the system and starts hosting any VMs. Since $N_{\text{over}}(t)$ is a counter of events from $t = 0$, it increases monotonically over time until a new server is added to the system. $N_{\text{over}}(t)$ is reset to 0, when a new server is added and starts hosting any VMs. In order to avoid any CPU overload event, additional server resource should be procured before $N_{\text{over}}(t)$ becomes positive. Let t_{proc} be the time when a server procurement decision is made. The ordered server will be delivered with a certain lead time *d*. After the delivery, the procured server becomes an unused server resource and UNP starts. Let t_{add} be the time when a procured server is added to the system and actually starts the operation. The UNP at the time t_{add} can be defined as the difference between t_{add} and the delivery time: $T_{\text{unp}}(t_{\text{add}}; t_{\text{proc}}) = t_{\text{add}} - (t_{\text{proc}} + d)$. The total cost until the new server is added to the system under the procurement decision at time t_{proc} is defined as below.

$$C(t_{\text{add}}; t_{\text{proc}}) = c_1 \cdot T_{\text{unp}}(t_{\text{add}}; t_{\text{proc}}) + c_2 \cdot N_{\text{over}}(t_{\text{proc}} + d)$$

where c_1 and c_2 are the cost coefficients. The notation $C(t_{\text{add}}; t_{\text{proc}})$ represents that the cost depends on t_{proc} while it is determined at t_{add}. We formulate the procurement decision problem by considering t_{proc} as a control variable that system administrator can determine.

Problem 6.1.1.1. *Server procurement decision problem*
Determine the optimal server procurement time t_{proc} which minimizes the total expected cost $C(t_{\text{add}}; t_{\text{proc}})$.

Note that $N_{\text{over}}(t_{\text{proc}} + d)$ is not given a priori since the number of overload events in the future depends on the dynamics of a system under uncertainty. In the following sections, we capture the system dynamics by stochastic processes to estimate the future CPU overload events and to describe an approach to determine the server procurement time t_{proc}.

6.1.1.4. System model

This section describes the models for VM demand arrivals and VM workload changes in a mobile thin-client service to capture the dynamics of the system. Based on the system model, the probability of new server demand caused by an expected CPU overload events is analyzed.

6.1.1.4.1. VM demand arrival

A new user requests an instantiation of a dedicated VM that can be accessed exclusively from user's mobile terminal. The demands of new VM instantiations can occur randomly independent of system state. Thus, VM demand arrivals can be modeled as an independent arrival process. Let H_s be the random variable representing the number of VM instantiation requests at the time period *s* (e.g., a day). The number of VMs in the system increases by H_s in the time period *s*. The arrival process may be governed by a specific stochastic process such as Poisson process or Markovian arrival process. Such an instance of stochastic process can be constructed from the real observation of demand arrival times. We examine the impact of different arrival processes in the evaluation section.

6.1.1.4.2. VM workload changes

The workloads of VMs in the system change depending on the usage of the applications by the users. Since each VM is dedicated for a specific user, the workload of the VM changes according to the application usage that is independent of other users activities. VM workload model needs to consider such usage pattern of mobile thin-client service.

To estimate the future workload in CPU utilizations, several data processing techniques are available for use, such as Kalman filter approaches [142], prediction using auto-regression models [143] and tendency-based method with polynomial fitting [144]. Such data processing techniques are useful to predict the continuous trend of CPU utilizations, while it is not suitable to model workload pattern in a specific time period, since it depends on the user activities in different time periods (e.g., morning or evening, holiday or weekday etc.). The workload is highly influenced by a user profile, which characterizes the user access patterns in individual time periods.

Based on the assumption that VM workload changes in a specific time period can be categorized into a certain small-set of patterns, we model the pattern-based model for VM workload changes. Let us define workload pattern as a vector of average CPU utilizations in K segmented time intervals, $\boldsymbol{u} = (u_1, u_2, \dots, u_K), \quad u_1, u_2, \dots, u_K \in [0,100]$. Workload changes in the consecutive periods can be represented by the transitions among workload patterns. Let S be the set of patterns, which is constructed from history data, and assume that any observation of workload change in a period is mapped onto a pattern $\boldsymbol{u}^{(i)} = \left(u_1^{(i)}, u_2^{(i)}, \dots, u_K^{(i)}\right), 1 \le i \le l,$ where l represents the number of patterns identified. By analyzing the statistics about the number of transitions among patterns $\boldsymbol{u}^{(i)}$ to $\boldsymbol{u}^{(j)}$ ($\boldsymbol{u}^{(i)}, \boldsymbol{u}^{(j)} \in S$), we can estimate the transition probabilities among individual patterns. The state space S and the estimated transition probabilities over S define a Discrete Time Markov chain (DTMC) which captures the probabilistic behavior of pattern transitions over the periods. Let $\boldsymbol{\Xi}$ be the $l \times l$ transition probability matrix of the DTMC whose (i, j)-element represents the transition probability from pattern $\boldsymbol{u}^{(i)}$ to $\boldsymbol{u}^{(j)}, 1 \le i, j \le l$. When the present workload pattern for a VM is identified, the DTMC characterized by $\boldsymbol{\Xi}$ can be used to predict the VM workload pattern in the next period.

6.1.1.4.3. Host demand probability

A new host is requested when the existing servers cannot accommodate the total expected workloads of all the hosted VMs. Both of the VM demand arrivals and workload changes affect the timing of a new host demand. Based on the abovementioned models, we analyze the probability of a new host demand at the time period t.

Let $n_{v,0}$ and $n_{h,0}$ be the number of VMs and the number of hosts in the system at present, respectively. The number of VMs at the end of the period t is given by

$$n_{v,t} = n_{v,0} + \sum_{s=1}^{t} H_s.$$

Provided that each host server has the same workload capacity C_h, the total capacity is represented by $C_h \cdot n_{h,0}$. A new host server will be requested when the total expected

workloads exceed the total capacity. Let $y\langle i,0\rangle \in \{1,\cdots,l\}$ be the index of workload pattern of VM *i* at present. By the workload estimation model, the probability vector of the workload of VM *i* in the time period *t* is given by $\boldsymbol{e}_{y\langle i,0\rangle}\boldsymbol{\Xi}^t$ where $\boldsymbol{e}_{y\langle i,0\rangle}$ is a $1\times l$ unit vector whose $y\langle i,0\rangle$-th element is 1. An estimation of the workload in *t* can be given by the weighted average of workload pattern $\boldsymbol{e}_{y\langle i,0\rangle}\boldsymbol{\Xi}^t\boldsymbol{U}$ where $\boldsymbol{U}$ is the pattern matrix defined by

$$\boldsymbol{U} = \begin{bmatrix} \boldsymbol{u}^{(1)} \\ \boldsymbol{u}^{(2)} \\ \vdots \\ \boldsymbol{u}^{(l)} \end{bmatrix} = \begin{bmatrix} u_1^{(1)} & u_2^{(1)} & \cdots & u_K^{(1)} \\ u_1^{(2)} & u_2^{(2)} & \cdots & u_K^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ u_1^{(l)} & u_2^{(l)} & \cdots & u_K^{(l)} \end{bmatrix}.$$

Similarly, an estimated workload of VM *j* that is added at time period $s, (1 \le s \le t)$ is given by $\boldsymbol{e}_{y\langle j,s\rangle}\boldsymbol{\Xi}^{t-s}\boldsymbol{U}$. Therefore, the total expected workload in the time period *t* is given by

$$\boldsymbol{Z}_t = (z_{1t}, z_{2t}, \ldots, z_{Kt}) = \sum_{i=1}^{n_{v,0}} \boldsymbol{e}_{y\langle i,0\rangle}\boldsymbol{\Xi}^t\boldsymbol{U} + \sum_{s=1}^{t}\sum_{j=1}^{H_s} \boldsymbol{e}_{y\langle j,s\rangle}\boldsymbol{\Xi}^{t-s}\boldsymbol{U}.$$

Since workload at any segment *k* in $\boldsymbol{Z}_t$ should be under the system available capacity, CPU overload event will occur when

$$\max_{1\le k\le K} z_{kt} > C_h \cdot n_{h,0}.$$

As a result, the probability that a new host server is requested in the time period *t* is

$$\Pr\left[\max_{1\le k\le K} z_{kt} > C_h \cdot n_{h,0}\right].$$

To compute the above probability, the workload patterns of all the servers, $y\langle i,0\rangle$ and $y\langle j,s\rangle$, need to be identified, while the randomness of VM request arrivals is also taken into consideration. For the sake of simplicity, we introduce an approximated model here. Since all the transitions of VM workload pattern are assumed to follow the DTMC, an observation of a specific workload pattern tends to follow the stationary probability after some time period. Denote the stationary probability as $\boldsymbol{\pi}$ which satisfies $\boldsymbol{\pi} = \boldsymbol{\pi}\boldsymbol{\Xi}$. By approximating $\boldsymbol{e}_{y\langle i,0\rangle}\boldsymbol{\Xi}^t \cong \boldsymbol{e}_{y\langle j,s\rangle}\boldsymbol{\Xi}^{t-s} \cong \boldsymbol{\pi}$, the total expected workload can be represented by

$$\boldsymbol{Z}_t = \left(n_{v,0} + \sum_{s=1}^{t} H_s \right) \boldsymbol{\pi U}.$$

Let $\boldsymbol{U}_k$ be the *k*-th column vector of $\boldsymbol{U}$ and then the server demand probability is

$$\Pr\left[\max_{1\le k\le K} z_{kt} > C_h \cdot n_{h,0}\right] = \Pr\left[\left(n_{v,0} + \sum_{s=1}^{t} H_s \right) \cdot \max_{1\le k\le K} \boldsymbol{\pi U}_k > C_h \cdot n_{h,0}\right]$$

$$= \Pr\left[\sum_{s=1}^{t} H_s > \frac{C_h \cdot n_{h,0}}{\max_{1\le k\le K} \boldsymbol{\pi U}_k} - n_{v,0}\right] = 1 - \Pr\left[\sum_{s=1}^{t} H_s \le \frac{C_h \cdot n_{h,0}}{\max_{1\le k\le K} \boldsymbol{\pi U}_k} - n_{v,0}\right].$$

When we assume a specific distribution for VM demand arrival process, the above probability can be computed by the probability of the number of VM arrivals until the end of the time period *t*.

6.1.1.5. Procurement planning framework

The host demand probability analyzed in the previous section can be used for deciding the server procurement time that may avoid any CPU overload events in the future. However, the analysis relies only on the external system behaviors such as VM demand arrivals and workload changes, and it does not take into account the internal system dynamics, namely VM resource reallocation mechanism in the cloud. Since the resource reallocation mechanism depends on the administrative policies and could have some variations, modeling the internal system behavior in a uniform way is not suitable. Therefore, instead of making a specific model for VM resource reallocation, we employ the framework that simulate a VM replacement algorithm based on the models introduced in the previous section.

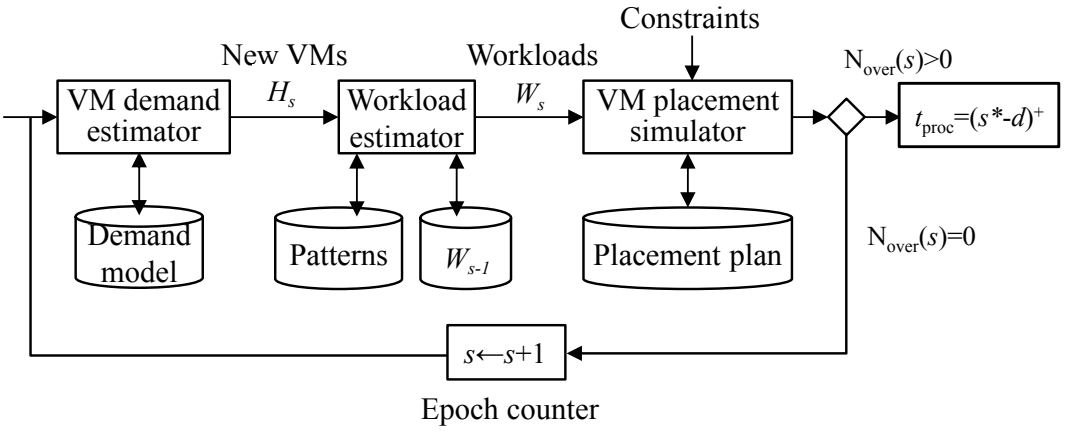


Figure 6.2 Block diagram of the server procurement decision framework [140]

6.1.1.5.1. Architecture

The overview architecture of the proposed framework is shown in Figure 6.2. The framework includes two estimators; VM demands estimator and workload estimator. The VM demand estimator predicts the number of new requests for VMs, H_s, in the time period s. On the other hand, the workload estimator is used for predicting the CPU workloads of individual VMs, W_s, for the period s. The estimated workloads are then fed into the VM placement simulator in which a VM placement algorithm determines the optimum VM placement in the server infrastructure under some allocation constraints. An important constraint is the host capacity constraint which ensures that the average CPU utilization should not exceed the specified host capacity. When the average CPU utilization exceeds the host capacity, CPU overload events occur and it is necessary to add a new server to the system.

To determine the effective server procurement decision time, the overall framework works as a simulator. We assume that VM replacement takes place in a periodic maintenance period (e.g. every midnight) and define *epoch* as the time period s bounded by the latest decision of VM placement. Unless stated, in the following content of the paper we regard an epoch as one-day period. For each epoch s, based on the estimated H_s and W_s, the VM replacement simulator determines whether the existing host servers are tolerant to the workload changes and increased demands. If any CPU overload events do not occur in the epoch s, the simulation proceeds to the next epoch by increment s. If not, it means CPU overload is not

evitable without additional server resource. Thus, a new server is necessary to start operation at this epoch. When we denote this time epoch as s^*, the server addition time is estimated by $t_{\mathrm{add}} = s^*$. Subtracting the lead time d from t_{add}, we can get the desirable server procurement time t_{proc}. In Figure 6.2, $(s^* - d)^+$ means $max(s^* - d, 0)$.

6.1.1.5.2. VM demand estimator

As described before, the number of VM demands in the epoch *s* can be estimated when we assume a specific arrival process. Poisson process is a simple example of the arrival process that has a constant arrival rate λ. Since H_s represents the number of VM demands in the time interval *T*, samples of H_s follow the Poisson distribution

$$\Pr[H_s = x] = \frac{(\lambda T)^x}{x!} \cdot e^{-\lambda T}.$$

The arrival rate of VM requests may not be constant but it alternately changes in non-deterministic time intervals. To deal with such a variable request arrival rate, Markovian Arrival Process (MAP) [145] could be employed for the arrival process whose arrival rate is governed by a Continuous Time Markov Chain (CTMC). If MAP is used as an underlying arrival process, VM demand estimator needs to have a memory to save the current phase of the MAP and the phase status is updated at every epoch. The parameter of MAP can be determined by fitting with empirical observations of inter-arrival times and their lag correlations. Moment matching method [146] and maximum likelihood estimation method [147] are two well-known approaches for MAP fitting.

6.1.1.5.3. VM workload estimator

VM workload estimator predicts the workloads of individual VMs by the model presented in the previous section. The DTMC characterized by the transition probability matrix $\boldsymbol{\Xi}$ is used to predict the VM workload pattern in the next epoch by the following steps.

(1) For a target VM, the latest workload pattern is observed as $\tilde{\boldsymbol{u}} = (\tilde{u}_1, \tilde{u}_2, \ldots, \tilde{u}_k)$. Compute the Euclid distances between $\tilde{\boldsymbol{u}}$ and any $\boldsymbol{u} \in S$ and determine the pattern $\boldsymbol{u}^{(i)}$ that has the shortest distance from $\tilde{\boldsymbol{u}}$.

(2) Determine a next workload pattern $\boldsymbol{u}^{(j)}, 1 \leq j \leq l$, which has a transition from $\boldsymbol{u}^{(i)}$ in the DTMC, sampled by the corresponding probability. In other words, $\boldsymbol{u}^{(j)}$ is chosen as the next workload pattern by probability $\boldsymbol{\Xi}_{ij}$.

We also need to estimate the workload of new VMs, which is expected to start in the epoch *s*. We can incorporate this boundary case into the pattern transition DTMC by adding the pattern for start-up workload. In the DTMC, the transition from start-up workload to any workload patterns is defined and the associated transition probabilities are estimated from the workload history of start-up VMs.

Instead of using DTMC, we may alternatively employ hidden Markov model (HMM) that includes hidden states with the DTMC. Such hidden states are particularly important if we construct both of workload patterns and pattern transition matrix from empirical data. Clustering approach to obtain the workload patterns from the observed data essentially involves estimation errors. From statistical point of view, it is reasonable to deal with the clustering and estimating transition probability matrix simultaneously. HMM is suitable for such cases and its parameters can be estimated by an expectation maximization (EM) algorithm. Since the design of the EM algorithm is beyond the scope of this chapter, we focus on the DTMC-based estimation approach in the following sections.

6.1.1.5.4. VM placement simulator

VM replacement simulator uses the estimated workloads and performs the simulation of VM replacement algorithm that is deployed in the data center. VM replacement algorithm, which reallocate VMs to host servers during a maintenance period, has a significant impact on the time to CPU overload event resulting in a new server addition. Depending on heuristic algorithms or optimization methods used in the system, the total number of VMs that can be hosted by the given host servers changes. Some heuristic bin-packing algorithms, such as first-fit decreasing (FFD) [145], are known to be useful for deriving a sub-optimal VM placement plan [149][150]. However, a simple bin-packing algorithm is not suitable to the case that VM reallocation is taken place periodically for adapting to workload changes. In VM replacement planning, it is important to minimize the number of VM migrations required

for moving to the next optimum placement because of the limited maintenance time window for VM reallocation.

There are several VM placement approaches that take into consideration the number of VM migrations required for replacement. A constraint programming approach like Entropy [151] uses a common Constraint Satisfaction Problem (CSP) solver to get a VM placement plan which can minimize the cost associated with reallocation while satisfying capacity and colocation constraints. CSP solution-based approach is very flexible to any constraints for VM placement planning, though it has the limited scalability and usually requires a long solution time. Since VM replacement simulator is essentially algorithm-agnostic, the framework can easily adapt to any kind of VM placement approaches including CSP solution-based approach. In the evaluation discussed in the next section, a heuristic VM replacement algorithm is considered.

6.1.1.6. Evaluation

To show the feasibility and the effectiveness of the model-based procurement decision, we conduct experimental studies based on the proposed framework with the real workload traces observed in a real system.

6.1.1.6.1. Workload characterization

The workloads of VMs in a mobile thin-client service are analyzed in [141]. The trace of CPU utilization of VMs in the system is used to construct typical workload patterns for workload estimation. From more than a hundred VMs, the raw CPU utilization traces are collected by one minute interval. From the raw data, we compute the average CPU utilizations for individual hours in a day to roughly capture the usage pattern.

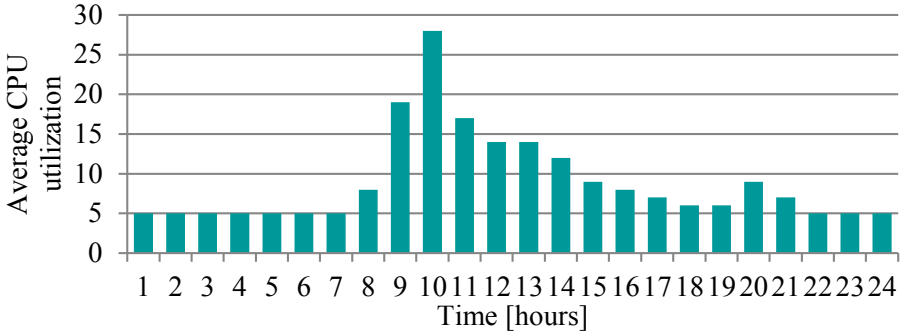


Figure 6.3 An example of monitored CPU workloads of a VM [140]

Figure 6.3 shows a representative pattern of a VM workload in a mobile thin-client service for enterprise use. Until 8 a.m. only base workloads, fixed amount of overhead of CPU utilization, are observed. Afterward the average CPU utilization reaches to a peak around 10 a.m. The amount of workloads gradually decreases in the afternoon to the base workload at the end of the day. The workload changes are affected by what applications are used in the VM (such as browser, mail client, media player and so on). Here we do not get into the details of the user applications, while we capture the workload characteristics in a way described in Section 6.1.1.4.2.

Based on the real workloads observed, we extract five representative workload patterns for evaluation; 1) VM start-up, 2) stable workload, 3) standard use pattern as shown in Figure 6.3, 4) VM is mainly used in the morning, and 5) in the afternoon. The pattern can be represented by a series of 1-hour average CPU utilizations from 7 a.m. to 22 p.m. The midnight hours (i.e., 23 p.m. to 6 a.m. tomorrow) are not considered in the pattern since the workload do not change significantly and the period might be used for maintenance. In the evaluation, we use the pattern matrix $\boldsymbol{U}$ and the pattern transition matrix $\boldsymbol{\Xi}$ defined below.

$$\boldsymbol{U} = \begin{bmatrix} \boldsymbol{u}^{(1)} \\ \boldsymbol{u}^{(2)} \\ \boldsymbol{u}^{(3)} \\ \boldsymbol{u}^{(4)} \\ \boldsymbol{u}^{(5)} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 24 & 8 & 6 & 6 & 5 & 5 & 5 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 8 & 19 & 28 & 17 & 14 & 14 & 12 & 9 & 8 & 7 & 6 & 6 & 9 & 7 \\ 4 & 4 & 10 & 8 & 5 & 5 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 34 & 19 & 11 & 8 & 6 & 6 & 5 \end{bmatrix},$$

$$\boldsymbol{\Xi} = \begin{bmatrix} 0 & 0.3 & 0.3 & 0.2 & 0.2 \\ 0 & 0.6 & 0.1 & 0.1 & 0.2 \\ 0 & 0.3 & 0.5 & 0.1 & 0.1 \\ 0 & 0.2 & 0.1 & 0.6 & 0.1 \\ 0 & 0.3 & 0.2 & 0 & 0.5 \end{bmatrix}.$$

The transition probabilities can be estimated from the statistics of pattern transitions as explained in Section 6.1.1.4.2. In this evaluation, however, we use a hypothetical transition matrix $\boldsymbol{\Xi}$ defined above due to the lack of sufficient data for estimation.

6.1.1.6.2. Demand arrival models

In the analyzed system [141], it is observed that 24 VMs were newly instantiated during a week, which means average VM demand arrival rate was roughly estimated as 0.14 VMs per hour. The demands for VMs do not arrive regularly and hence we capture the demand arrival process by MAP. The background CTMC of MAP is characterized by the infinitesimal generator matrix $\boldsymbol{D}$. $\boldsymbol{D}$ is decomposed into two matrices $\boldsymbol{D}_0$ and $\boldsymbol{D}_1$ (i.e., $\boldsymbol{D} = \boldsymbol{D}_0 + \boldsymbol{D}_1$), where $\boldsymbol{D}_0$ represents the phase transition without arrival and $\boldsymbol{D}_1$ represents the transitions with an arrival. We consider the following three different instances of MAP.

(1) Poisson process

When we set $\boldsymbol{D}_0 = -\lambda$ and $\boldsymbol{D}_1 = \lambda$, the MAP is a Poisson process with parameter λ.

(2) Switched Poisson process

The MAP with the matrices

$$\boldsymbol{D}_0 = \begin{bmatrix} -(\sigma_1 + \lambda_1) & \sigma_1 \\ \sigma_2 & -(\sigma_2 + \lambda_2) \end{bmatrix}, \quad \boldsymbol{D}_1 = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

is known as a switched Poisson process where the arrival rate changes by the phase (1 or 2). The parameters σ_1 and σ_2 govern the phase transitions (i.e., switching rate), while λ_1 and λ_2 represent the arrival rates in the individual phases.

(3) Burst arrival

One of the representations of bursty arrival process is given by the MAP with the matrices

$$\boldsymbol{D}_0 = \begin{bmatrix} -(\gamma_1 + \gamma_2) & 0 \\ \mu & -\mu \end{bmatrix}, \quad \boldsymbol{D}_1 = \begin{bmatrix} \gamma_1 & \gamma_2 \\ 0 & 0 \end{bmatrix}$$

where many arrivals are expected by high arrival rate in phase 1 and no arrivals in phase 2. The parameter μ represents the transition rate from phase 2 to phase 1, which governs the time between consecutive burst arrival periods.

We determine the parameter values for the above three models shown in Table 6.1 so that the average arrival rate coincides with the same value.

Table 6.1 Parameter values for demand arrival models

Model	Parameter	Values [1/hours]
Poisson process	λ	1/12
Switched Poisson process	λ_1	1/24
	λ_2	1/8
	σ_1, σ_2	1/72
Burst arrival	γ_1	1/4
	γ_2	1/24
	μ	1/60

6.1.1.6.3. Comparative targets

Besides the simulation approach presented in Section 6.1.1.5, the black box modeling approach and the two heuristic approaches are used as the comparative targets.

- Conservative policy: Under this policy, the server procurement is determined to prepare a backup server all the time so as to minimize the CPU overload events. The approach is preferable when CPU overloads induce a serious consequence in user-perceived performance. Until a new server is requested in response to CPU overload event, the backup server remains inactive. Immediately after activating the backup server, the next order of a new server is placed. The ordered server is delivered after the delivery lead-time and it becomes a new backup server. The period a backup server remains inactive is counted as UNP.

- Reactive policy: The reactive policy decides server procurements reactively after observing any CPU overload events in the system. The ordered server is added in the system after the delivery. Although the system may encounter further overload events during the wait time for new server delivery, the reactive policy can avoid unnecessary server stock and hence can minimize the cost of UNP.

- Black box modeling approach: Based on the models presented in Section 6.1.1.4.3, this method estimates the probability of new server demands only from the estimations of external system behaviors. Server procurement is determined when the estimated probability exceeds a certain threshold value. In our experiments, we set this threshold value to 0.5. Since the method does not take into account the dynamics of VM replacement in the cloud system, we refer to this method as black box modeling approach.

6.1.1.6.4. Comparison of the total costs

To evaluate the effectiveness of the proposed framework, first we conducted the comparative evaluations with four different decision methods under the same VM demand sequence and VM workload changes. The VM demand sequence is generated by the Poisson process described in Section 6.1.1.6.2. For the VM placement algorithm, we assume that a heuristic algorithm [152] that attempts to minimize the number of VM migrations is used and the same algorithm is used in the simulation. The workload estimation is based on the patterns

constructed in Section 6.1. The host capacity C_h is set to 600 that represents the limit of the average CPU utilization in the same time interval. The lead-time of server delivery is set to three days (d=3). The simulation is conducted for 200 days and the total cost is computed from the observed CPU overloads and UNPs. The cost coefficients c_1 and c_2 are set to 0.7 and 0.3, respectively. Figure 6.4 shows the box chart that summarizes the results of ten simulated experiments for four approaches. As can be seen, the simulation approach achieves the minimum cost compared with other approaches. In the best case, the total expected cost can be reduced by up to 71.1% and 57.1% in comparison with conservative policy and reactive policy, respectively.

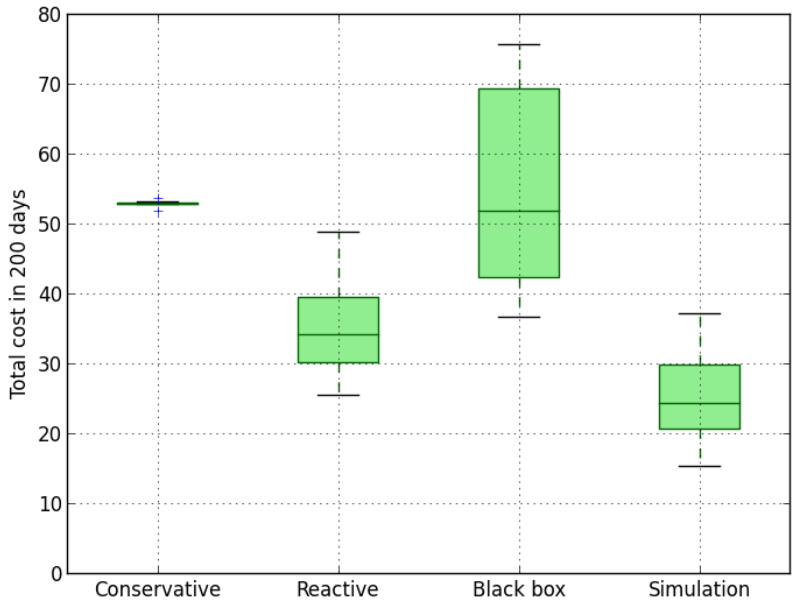


Figure 6.4 Comparison of the total costs by four difference methods [140]

As defined in the cost function in Section 6.1.1.3, the UNP and the CPU overload events are the two cost components. The average UNP and CPU overloads observed in the above experiments are shown in Figure 6.5.

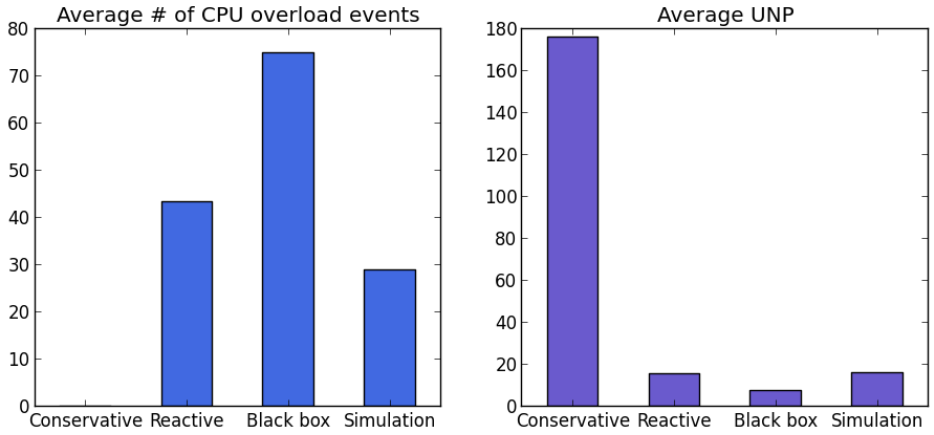


Figure 6.5 Comparison of the average number of CPU overloads and UNP [140]

As expected, conservative policy never experiences the CPU overload events (in the left chart), although almost all the time there is an unused backup server that is the cause of the largest UNP (in the right chart). Both the reactive policy and black box modeling approach encounter relatively a large number of CPU overload events due to the delayed decision of server procurement. Compared with those approaches, the proposed simulation method provides the most balanced solution.

In order to examine the trend of cost factors, we carry out the simulation for one-year period and see the trend of CPU overloads and UNPs. Figure 6.6 shows the simulation result.

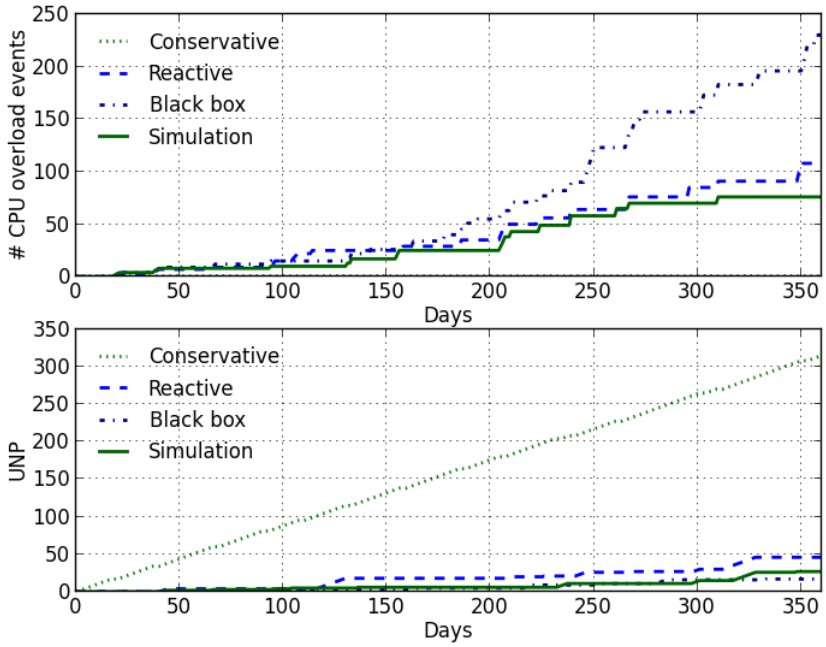


Figure 6.6 The trend of CPU overloads and UNP for one-year simulation [140]

As time goes by, the conservative policy increases UNP, while the reactive policy increases the number of CPU overload events. Although the black-box modeling approach achieves good performance in the early period, it encounters a large number of CPU overloads, especially after 200 days. This phenomenon is caused by the fact that the black-box modeling approach does not take into account the impact of VM replacement algorithm.

6.1.1.6.5. Impacts of arrival process

We conduct further simulation experiments by varying the arrival models described in Section 6.1.1.6.2. Figure 6.7 and Figure 6.8 show the results of the same experiments by replacing the arrival model to switched Poisson process and burst arrival, respectively. To clarify the difference of the results of assumed arrival process, the simulation results by Poisson arrival process is also compared.

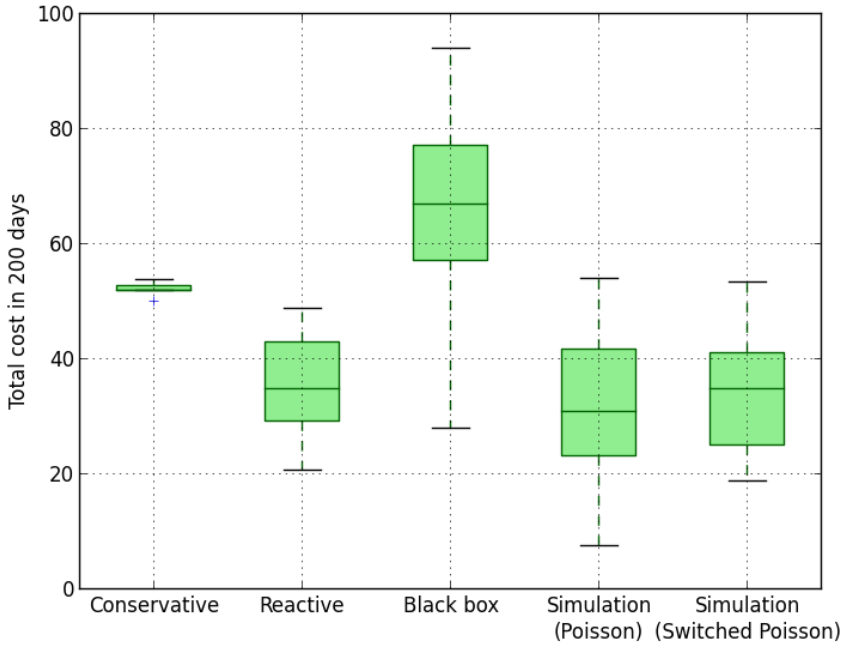


Figure 6.7 Simulation results by switched Poisson process [140]

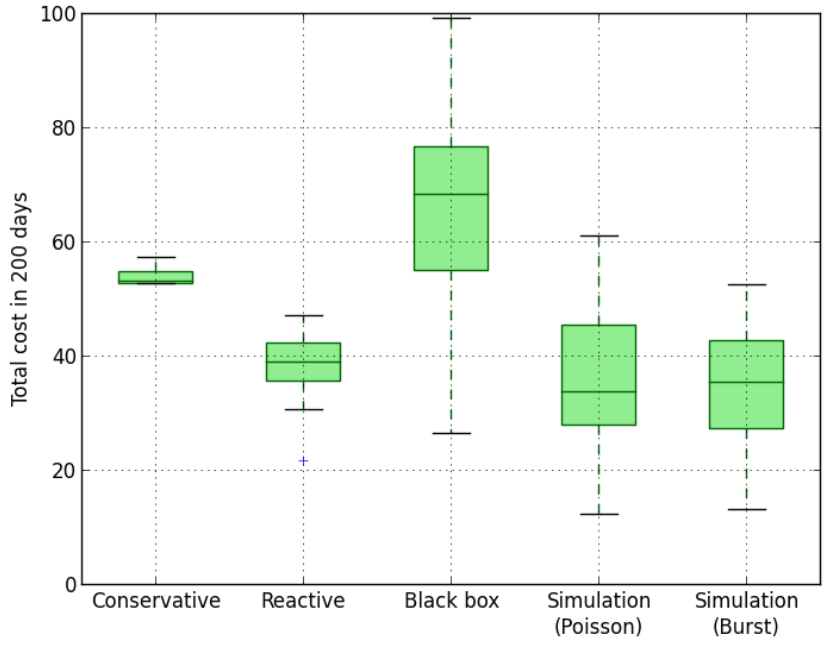


Figure 6.8 Simulation results by burst arrival [140]

Against the expectation, the total costs of the simulation approach by switched Poisson process and burst arrival model do not provide significant advantages over the simulation

with simple Poisson process by the 200 days of experiments. Although the framework can easily adopt different VM demand arrival models, the impact of the arrival process on the total cost is not so significant in this experimental setting.

6.1.1.7. Related work

Mobile thin-client service is considered as a type of mobile cloud computing that combines the technologies for cloud computing and mobile communications to enable a rich mobile computing [139]. Mobile computation offloading using rich cloud resources is one of the major issues discussed in this research field, while resource management in cloud is more critical in thin-client system. An architecture of mobile thin-client computing is discussed in [153], where cloud resource management optimizes the number of users in hosts to minimize the energy cost in the cloud. Our work differs from them in terms of the problem definition that intends to optimize server procurement decisions.

An important motivation to introduce thin-client system in companies is protecting data from unsecure terminals and networks. There is, however, a trade-off between the security and user-perceived performance. WriteShield [154] is presented as a pseudo thin-client system in which only write-protected data is stored in clients so that OS and applications in the client can use local resources. To analyze the performance of thin-client system using cloud, VDBench [155] provides an efficient benchmarking toolkit. A quality of experience (QoE) metric for thin-client service is defined and evaluated in [156]. These works can complement our work to improve the user experience while keeping adequate level of security.

Workload characterization is an important process for optimizing resources in cloud and datacenter systems. In [157], the authors collected and analyzed the CPU and memory usage of desktop computers to characterize the aggregated workloads of desktop cloud. In [158], the workload characterization of enterprise applications in data centers was studied for predicting future demands and capacity planning. The studies confirm that the characterized patterns of workloads are beneficial to predict the workload changes for resource optimization. In this work, we exploit stochastic models to capture such pattern-based

workload changes that are suitable for simulating periodic VM replacements in our framework.

6.1.1.8. Summary

Timely decision of server procurement to private cloud for mobile thin-client service is an important and challenging issue. We formulated the problem of server procurement decision in light of the trade-off between the cost of server unused period caused by a premature decision and the cost associated with service level violation due to delayed decision. To find the solution to the problem, the procurement planning framework is introduced which predicts the time of resource contention by estimating VM demands, VM workloads and the behavior of resource allocation mechanism in the cloud. Through the simulation studies, we confirm that the combination of stochastic process capturing the external system behavior with the internal resource management simulation enables a better estimation of server procurement decision in terms of the total operational cost. The suggested problem can also be considered as a sequence of optimal stopping problems when we regard each server procurement is one-shot decision problem. The analysis of a sequence of optimal stopping problems for server procurement decision could be discussed in the future work.

Chapter 7

Conclusions and future research

Dependability of IT systems is crucial for our society and economy. It is a fundamental research challenge to continuously ensure and improve the dependability of IT systems that are changing, evolving, and connecting with other systems. We should understand comprehensively the configuration and behavior of target systems, analyze the preference of their configuration and operations in a quantitative manner, and explore potential means to improve their performance and availability. To support such a process, the stochastic models discussed in this thesis are very powerful tools that can deal with uncertainties, such as component failures, and evaluate system designs quantitatively. Based on numerous existing studies from the research community of dependable computing, software reliability engineering, network system management etc., this thesis presented our contributions to these research domains. The specific contributions discussed in Chapters 4, 5, and 6 are summarized below.

Chapter 4 addressed the software aging problem and showed the effectiveness of software rejuvenation and life-extension. In Section 4.2.2, the optimal policy for dynamic software rejuvenation in a job processing system affected by service degradation was presented. We formulated the problem as an optimal stopping problem and analytically derived the optimal condition to perform software rejuvenation. In this study, it was assumed that the newly arrival jobs were rejected during the decision process. Thus, in Section 4.2.3, by relaxing the assumption, we reformulated the optimal stopping problem for rejuvenation decision to a degraded job processing system. The analytical results showed that the relation among the cost of delayed jobs, that of dropped jobs due to rejuvenation, and the traffic intensity are key to determining the policy, while the number of queued jobs in the system has no impact.

This gives **a quite simple guideline for determining rejuvenation action**. Subsequently, the effectiveness of software life-extension was investigated. Section 4.3.1 presented some experimental results of software life-extension for a memcached system. **The effectiveness of software life-extension was experimentally evaluated** in terms of system availability, the number of request drops, and cache hit rate of the memcached. In the subsequent study, Section 4.3.2 presented a semi-Markov model for software life-extension to analyze the optimal schedule for life-extension. **We showed the condition under which there exists a unique optimal interval to trigger software life-extension** in terms of steady-state system availability. Moreover, the hybrid approach where software life-extension is followed by rejuvenation was presented. **Through numerical experiments, we showed that the hybrid approach achieved the highest system availability** among the other individual approaches in which software life-extension or rejuvenation is performed exclusively.

Chapter 5 focused on the data availability issue. In Section 5.2.1, for analyzing the availability and performability of RAID storage systems, disk failures and subsequent rebuild operations are modeled by a Markov regenerative process (MRGP). While we confirmed that the modeling error in traditional CTMC models did not have a significantly impact data availability, it must have a non-trivial impact on performability evaluation. **The analytical solution of MRGP gives an accurate estimate of RAID storage performability.** In Section 5.2.2, an approach to derive the optimal data backup schedule for multiple data sets in a system was presented. **The optimal choice of backup actions (either full backup, partial backup, or skip) in terms of data availability can be obtained as the solution of Markov decision process.** The presented algorithm is useful to construct an instance of MDP from the requirements for data recovery objectives (i.e., RPO and RTO).

Chapter 6 presented a framework for guiding a server procurement decision for avoiding potential performance problems in a private cloud system for a mobile thin-client service. The framework is based on VM demand estimation, VM workload estimation, and resource reallocation simulator to predict the demand for a new server. In particular, DTMC based workload estimation method was presented. Through simulation experiments, **we confirmed**

that the proposed framework can reduce the total cost by up to 71% compared with a heuristic approach preparing a standby server all the time.

While this thesis mainly focused on availability, performance, and performability of software systems, security and safety are other important perspectives of system dependability. Ensuring system security is increasingly important as threats of cyber-attacks become prominent worldwide. It is interesting to see that software rejuvenation has also been applied for improving system security by recovering the compromised software execution environment [159]. Analyzing the impact of maintenance operations on security as well as performability is an important research issue for dependable computing systems. Safety engineering is also a challenging research domain because recently any outages of IT systems can cause serious problems on human lives and societies [160]. Ensuring safety may require a system-thinking approach rather than a reliability modeling approach, which is largely based on the concept of reductionism. To not violate safety conditions, several safety engineering methodologies, such as HAZOP [161] and STPA [162], focus on finding hazardous situations using guide words. Since recent IoT systems affecting the real world need to address safety concerns as well as system availability, an integrated dependability engineering approach is necessary. Further investigations on the intersections of availability, security, and safety of systems are important toward dependable IT systems and societies.

Bibliography

[1] A. Avizienis, J.C. Laprie, B. Randell and C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, No. 1, 2004.

[2] Amazon EC2 Service Level Agreement, September 2017.
URL https://aws.amazon.com/ec2/sla/

[3] Uptime Institute, Data Center Site Infrastructure Tier Standard: Topology, 2012.

[4] Colocation America, Data Center Standards (Tiers I-IV), September 2017.
URL https://www.colocationamerica.com/data-center/tier-standards-overview.htm

[5] P. E. Heegaard and K. S. Trivedi, Network survivability modeling, Computer Networks, Vol. 53, No. 8, pages 1215–1234, 2009.

[6] ANSI T1A1.2 Working Group on Network Survivability Performance, Technical report on enhanced network survivability performance, ANSI, Tech. Rep. TR No. 68, February 2001.

[7] P. E. Heegaard, B. E. Helvik, K. S. Trivedi, and F. Machida, Survivability as a Generalization of Recovery, In *International Conference on the Design of Reliable Communication Networks (DRCN)*, pages 133-140, 2015.

[8] J. C. Laprie, From dependability to resilience, In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2008.

[9] R. Ghosh, D. Kim, and K. S. Trivedi, System resiliency quantification using non-state-space and state-space analytic models, Reliability Engineering & System Safety, vol. 116, pp. 109-125, 2013.

[10] H. Okamura, J. Guan, C. Luo, and T. Dohi, Quantifying resiliency of virtualized system with software rejuvenation, *IEICE Trans. on Fundamentals of Elect., Comm. and Comp. Sciences,* Vol.E98-A, No. 10, pages 2051-2059, 2015.

[11] F. Machida, M. Fujiwaka, S. Koizumi, and D. Kimura, Optimizing resiliency of distributed video surveillance system for safer city, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 17-20, 2015.

[12] F. Machida, D. Kim, and K. S. Trivedi, Modeling and analysis of software rejuvenation in a server virtualized system with live VM migration, *Performance Evaluation*, vol. 70, no. 3, pages 212-230, 2013.

[13] J. F. Meyer, On evaluating the performability of degradable computing systems, *IEEE Transactions on Computers*, vol. 29, no. 8, pages 720-731, Aug, 1980.

[14] R. Huslende, A combined evaluation of performance and reliability for degraded systems, *ACM/SIGMETRICS Performance Evaluation Review*, vol. 10, no. 3, pages 157-164, 1981.

[15] R. M. Smith, K. S. Trivedi, and A. Ramesh, Performability analysis: measures, an algorithm and a case study, *IEEE Trans. on Computers*, vol. C-37, no. 4, pages 406-417, Apr, 1988.

[16] D. Logothetis, K. S. Trivedi, The Effect of Detection and Restoration Times for Error Recovery in Communication Networks, *Network and Systems Management*, Vol. 5 No. 2, pages 173-195, 1997.

[17] B. Haverkort, R. Marie and G. Rubino, and K. S. Trivedi, Performability modeling tools and techniques, *John Wiley & Sons*, Chichester, England, 2001.

[18] K. S. Trivedi, E. Andrade, F. Machida, Combining performance and availability analysis in practice, *Advances in Computers*, vol. 84, pages 1-38, 2012.

[19] K. S. Trivedi and A. Bobbio, Reliability and availability engineering : modeling, analysis and applications, *Cambridge University Press*, 2017.

[20] J. B. Dugan, S. Bavuso, and M. Boyd, Dynamic fault tree models for fault tolerant computer systems, *IEEE Trans. on Reliability*, vol. 41, no. 3, pages 363–377, 1992.

[21] S. Distefano and L. Sing, A new approach to modeling the system reliability: dynamic reliability block diagrams, In *Reliability and Maintainability Symposium (RAMS)*, pages. 189-195, 2006.

[22] M. R. Lyu, Handbook of software reliability engineering, McGraw-Hill, Inc., 1996.

[23] V. G. Kulkarni, Modeling and analysis of stochastic systems, *CRC Press*, 2016.

[24] R. E. Bellman, Dynamic Programming, *Princeton University Press*, 1957.

[25] M. L. Puterman, Markov decision processes: discrete stochastic dynamic programming, *John Wiley & Sons*, 2014.

[26] M. Grottke and K. S. Trivedi, Fighting bugs: Remove, retry, replicate and rejuvenate, *IEEE Computer*, vol. 40, no. 2, pages 107-109, 2007.

[27] M. Grottke, R. Matias Jr., and K. S. Trivedi, The fundamentals of software aging, In *International Workshop on Software Aging and Rejuvenation (WoSAR2008)*, pages 1-6, 2008.

[28] M. Grottke, A. P. Nikora, and K. S. Trivedi, An empirical investigation of fault types in space mission system software, In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, pages 447-456, 2010.

[29] F. Machida, J. Xiang, K. Tadano and Y. Maeno, Aging-related bugs in cloud computing software, In *International Workshop on Software Aging and Rejuvenation (WoSAR)*, pages 287-292, 2012.

[30] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, Software aging analysis of the Linux operating system, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 71-80, 2010.

[31] D. Cotroneo, F. Fucci, A. K. Iannillo, R. Natella, R. Pietrantuono, Software aging analysis of the Android mobile OS, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 478-489, 2016.

[32] G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *IEEE Aerospace Conference*, pages 4186–4199, 2005.

[33] S. Ballerini, L. Carnevali, M. Paolieri, K. Tadano, and F. Machida, Software rejuvenation impacts on a phased-mission system for Mars exploration, In *International Workshop on Software Aging and Rejuvenation (WoSAR)*, pages 275-280, 2013.

[34] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi, A methodology for detection and estimation of software aging, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 283-292, 1998.

[35] F. Machida, J. Xiang, K. Tadano, Y. Maeno, Combined server rejuvenation in a virtualized data center, In *IEEE International Conference on Autonomic & Trusted Computing (ATC)*, pages 486-493, 2012.

[36] D. Cotroneo and R. Natella, Monitoring of aging software systems affected by integer overflows, In *International Workshop on Software Aging and Rejuvenation (WoSAR)*, pages 265-270, 2012.

[37] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, A survey of software aging and rejuvenation studies, *ACM Journal on Emerging Technologies in Computing Systems*, vol. 10, no. 1, 8, 2014.

[38] R. Matias Jr., K. S. Trivedi, and P. R. M. Maciel, Using accelerated life tests to estimate time to software aging failure, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 211-219, 2010.

[39] R. Matias Jr., Pedro A. Barbetta, K. S. Trivedi, and P. J. Freitas Filho, Accelerated degradation tests applied to software aging experiments, *IEEE Transactions on Reliability*, vol. 59, no. 1, pages 102-114, 2010.

[40] F. Machida, R. Matias Jr. and A. Andrzejak, On the effectiveness of Mann-Kendall test for detection of software aging, In *International Workshop on Software Aging and Rejuvenation (WoSAR)*, pages 269-274, 2013.

[41] R. Matias Jr., A. Andrzejak, F. Machida, D. Elias and K. S. Trivedi, A systematic differential analysis for fast and robust detection of software aging, In *IEEE Reliable and Distributed Systems (SRDS)*, pages 311-320, 2014.

[42] F. Langner, and A. Andrzejak, Detecting software aging in a cloud computing framework by comparing development versions, In *IFIP/IEEE Integrated Network Management (IM)*, pages 896-899, 2013.

[43] M. Grottke and B. Schleich, How does testing affect the availability of aging software systems?, *Performance Evaluation*, vol. 70, no.3, pages 179-196, 2013.

[44] Y. Huang, C. Kintala, N. Kolettis, N. D. Fulton, Software rejuvenation: analysis, module and applications, In *IEEE International Symposium on Fault Tolerant Computing (FTCS-25)*, pages 381–390, 1995.

[45] J. Alonso, R. Matias Jr., E. Vicente, A. Maria, and K. S. Trivedi, A comparative experimental study of software rejuvenation overhead, In *Performance Evaluation*, Vol. 70, pages 231–250, 2013.

[46] H. Okamura and T. Dohi, Optimization of opportunity-based software rejuvenation policy, In *International Workshop on Software Aging and Rejuvenation (WoSAR)*, pages 283-286, 2012.

[47] H. Okamura and T. Dohi, Optimal trigger time of software rejuvenation under probabilistic opportunities, *IEICE Transactions on Information and System* Vol. E96-D, No. 9, 2013.

[48] F. Machida, J. Xiang, K. Tadano and Y. Maeno, Software life-extension: a new countermeasure to software aging, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 131-140, 2012.

[49] F. Machida and N. Miyoshi, An optimal stopping problem for software rejuvenation in a job processing system, In *International Workshop on Software Aging and Rejuvenation (WoSAR)*, pages 139-143, 2015.

[50] F. Machida, and N. Miyoshi, Analysis of an optimal stopping problem for software rejuvenation in a deteriorating job processing system, *Reliability Engineering and System Safety*, vol. 168, pages 128-135, 2017.

[51] F. Machida, and P. Maciel, Chapter 8 Markov chains and Petri nets, In *Handbook of Software Aging and Rejuvenation*, submitted.

[52] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, Analysis of software rejuvenation using Markov regenerative stochastic Petri nets, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 180-187, 1995.

[53] T. Dohi, K. Goševa-Popstojanova, and K. S. Trivedi, Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule, In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 77-84, 2000.

[54] T. Dohi, K. Goševa-Popstojanova, and K. S. Trivedi, Estimating software rejuvenation schedules in high assurance systems, *Computer Journal*, vol. 44, no. 6, pages 473-482, 2001.

[55] W. Xie, Y. Hong, K.S. Trivedi, Analysis of a two-level software rejuvenation policy, *Reliability Engineering and System Safety*, Vol. 87, pages 13-22, 2005.

[56] D. Chen, and K.S. Trivedi, Analysis of periodic preventive maintenance with general system failure distribution, In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 103-107, 2001.

[57] Y. Bao, X. Sun, and K.S. Trivedi, A workload-based analysis of software aging and rejuvenation, *IEEE Trans. on Reliability*, Vol. 54, No. 3, pages 541-548, 2005.

[58] S. Garg, S. Pfening, A. Puliafito, M. Telek, and K. S. Trivedi, Analysis of preventive maintenance in transactions based software systems, *IEEE Transactions on Computers*, vol.47, pages 96-107, 1998.

[59] K. Vaidyanathan, and K. S. Trivedi, A comprehensive model for software rejuvenation, *IEEE Transactions on Dependable and Secure Computing*, Vol. 2, No. 2, pages 124-137, 2005.

[60] J. Zhao, Y. Jin, K. S. Trivedi, and R. Matias Jr., Injecting memory leaks to accelerate software failures, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 260-269, 2011.

[61] J. Zhao, Y. Wang, G. Ning, K. S. Trivedi, R. Matias Jr., and K. Y. Cai, A comprehensive approach to optimal software rejuvenation, *Performance Evaluation*, vol. 70, no. 11, pages 917-933, 2013.

[62] S. M. Ross, Introduction to Probability Models, 7th ed., *Academic Press*, 2000.

[63] F. Machida, V. F. Nicola, and K. S. Trivedi, Job completion time on a virtualized server with software rejuvenation, *ACM Journal on Emerging Technologies in Computing Systems*, vol. 10, no. 1, 10, 2014.

[64] V. G. Kulkarni, V. V. Nicola and K. S. Trivedi, The completion time of a job on multimode systems, *Advances in Applied Probability*, vol. 19, pages 932-954, 1987.

[65] A. Bobbio, S. Garg, M. Gribaudo, A. Horvath, M. Sereno, M. Telek, Modeling software systems with rejuvenation, restoration and checkpointing through fluid stochastic Petri nets, In *International Workshop on Petri Nets and Performance Models (PNPM)*, pages 82-91, 1999.

[66] K. Vaidyanathan, R.E. Harper, S.W. Hunter, K.S. Trivedi, Analysis and implementation of software rejuvenation in cluster systems, In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 62–71, 2001.

[67] V. Castelli, R.E. Harper, P. Heidelberger, S.W. Hunter, K.S. Trivedi, K. Vaidyanathan, W.P. Zeggert, Proactive management of software aging, *IBM Journal of Research and Development*, Vol. 45, No.2, pages 311-332, 2001.

[68] G. Ciardo, A. Blakemore, P.F. Chimento, J.K. Muppala, and K.S. Trivedi, Automated generation and analysis of Markov reward models using stochastic reward nets, *Linear Algebra, Markov Chains and Queuing Models*, vol. 48, Springer, pages 145-191, 1993.

[69] D. Wang, W. Xie, and K. S. Trivedi, Performability analysis of clustered systems with rejuvenation under varying workload, *Performance Evaluation*, vol. 64, no. 3, pages 247-265, 2007.

[70] K. S. Trivedi, Probability and Statistics with Reliability, Queuing, and Computer Science Applications, *John Wiley & Sons*, 2001.

[71] F. Machida, D. Kim, and K. S. Trivedi, Modeling and analysis of software rejuvenation in a server virtualized system, In *International Workshop on Software Aging and Rejuvenation (WoSAR)*, pages 1-6, 2010.

[72] M. Melo, P. Maciel, J. Araujo, R. Matos, and C. Araujo, Availability study on cloud computing environments: Live migration as a rejuvenation mechanism, In *International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV)*, pages 1-6, 2013.

[73] K. S. Trivedi and R. Sahner, SHARPE at the age of twenty two, *SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pages 52-57, 2009

[74] R. German, C. Kelling, A. Zimmermann, G. Hommel, TimeNET: a toolkit for evaluating non-Markovian stochastic Petri nets, *Performance Evaluation*, vol. 24, pages 69–87, 1995.

[75] J. Rumbaugh, I. Jacobson, and G. Booch, Unified modeling language reference manual, *Pearson Higher Education*, 2004.

[76] S. Friedenthal, A. Moore, and R. Steiner, A practical guide to SysML: systems modeling language, *Morgan Kaufmann*, 2014.

[77] F. Machida, E. Andrade, D. Kim, K. S. Trivedi, Candy: component-based availability modeling framework for cloud service management using SysML, In *IEEE International Symposium on Reliable and Distributed Systems (SRDS)*, pages 209-218, 2011.

[78] E. Andrade, F. Machida, D. Kim, and K. Trivedi, Modeling and analyzing server system with rejuvenation through SysML and stochastic reward nets, In *International Conference on Availability, Reliability and Security (ARES)*, pages 22-26, 2011.

[79] D. Chen, and K. S. Trivedi, Optimization for condition-based maintenance with semi-Markov decision process, *Reliability Engineering & System safety*, Vol.90, pages 25-29, 2005.

[80] H. Eto and T. Dohi, Determining the Optimal Software Rejuvenation Schedule via Semi-Markov Decision Process, *Computer Science*, Vol. 2, No. 6, pages 528-535, 2006.

[81] S. Pfening, S. Garg, A. Puliafito, M. Telek and K.S. Trivedi, Optimal rejuvenation for tolerating soft failure, *Performance Evaluation*, 27/28, pages 491-506, 1996.

[82] H. Okamura and T. Dohi, Dynamic software rejuvenation policies in a transaction-based system under Markovian arrival processes, *Performance Evaluation*, Vol 70. No. 3, 2013.

[83] F. Machida, J. Xiang, K. Tadano and Y. Maeno, Lifetime extension of software execution subject to aging, *IEEE Transactions on Reliability*, Vol. 66, No. 1, pages 123-134, 2017.

[84] K. Vaidyanathan and K. S. Trivedi, A measurement-based model for estimation of resource exhaustion in operational software systems, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 84-93, 1999.

[85] A. Bovenzi, D. Cotroneo, R. Pietrantuono, S. Russo, Workload characterization for software aging analysis, In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 240-249, 2011.

[86] K. Kourai, Fast and correct performance recovery of operating systems using a virtual machine monitor, In *International Conference on Virtual Execution Environments (VEE11)*, pages 99-110, 2011.

[87] K. S. Trivedi, R. K. Mansharamani, D. Kim, M. Grottke, M. Nambinar, Recovery from Failures Due to Mandelbugs in IT Systems, In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 224-233, 2011.

[88] P. K. Sen, Estimates of the regression coefficient based on Kendall's tau, *Journal of the American Statistical Association*, Vol. 63, No. 4, pages 1379-1389, 1968.

[89] B. Schroeder, A. Wierman, and M. Harchol-Balter, Open versus closed: a cautionary tale, In *Network System Design and Implementation (NSDI)*, pages 239-252, 2006.

[90] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T, Leu, and W. S. Beebee Jr., Enhancing server availability and security through failure-oblivious computing, In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[91] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, Interpreting the data: Parallel analysis with Sawzall, *Scientific Programming*, Vol. 13, No. 4, pages 277-298, 2005.

[92] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, Rx: Treating bugs as allergies: A safe method to survive software failures, In *Symposium on Operating Systems Principles (SOSP)*, pages 235-248, 2005.

[93] G. Barish and K. Obraczka. World wide web caching: Trends and techniques. *IEEE Communications Magazine*, May 2000.

[94] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, Dynamo: amazon's highly available key-value store, In *Symposium on Operating Systems Principles (SOSP)*, pages 205-220, 2007.

[95] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, Pnuts: Yahoo!'s hosted data serving platform, In *VLDB Endowment*, Vol. 1, No. 2, pages 1277-1288, 2008.

[96] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, Fast crash recovery in RAMCloud, In *Symposium on Operating Systems and Principals (SOSP)*, pages 29-41, 2011.

[97] A. Mallet, Numerical Inversion of Laplace Transform, in Wolfram Library Archive, http://library.wolfram.com/infocenter/MathSource/2691/, 2000.

[98] D. Kececioglu, Reliability Engineering Handbook (Vol. 1 and 2). *Prentice-Hall, Inc.*, 1991.

[99] F. Machida, R. Xia and K. S. Trivedi, Performability modeling for RAID storage systems by Markov regenerative process, *IEEE Transactions on Dependable and Secure Computing*, Vo. PP, No. 99, 2015.

[100] B. Schroeder and G. A. Gibson, Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?, In *USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[101] J. G. Elerath and M. Pecht, Enhanced Reliability Modeling of RAID Storage Systems, In *IEEE/IFIP International Conference on Dependable Computing and Networks (DSN)*, pages 175-184, 2007.

[102] J. G. Elerath, A simple equation for estimating reliability of an N+1 redundant array of independent disks (RAID), In *IEEE/IFIP International Conference on Dependable Computing and Networks (DSN)*, pages 484-493, 2009.

[103] E. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, KK Rao, P. Zhou, Evaluating the impact of undetected disk errors in RAID systems, In *IEEE/IFIP International Conference on Dependable Computing and Networks (DSN)*, pages 484-493, 2009.

[104] K. M. Greenman, J. S. Plank, and J. J. Wylie, Meantime to meaningless: MTTDL, Markov models, and storage system reliability, In *USENIX Workshop on Hot Topics in Storage and File Systems*, pages 1-5, 2010.

[105] J. G. Elerath and J. Schindler, Beyond MTTDL: A Closed-Form RAID 6 Reliability Equation, *ACM Transactions on Storage*, Vol. 10, No. 2, 2014.

[106] V. Venkastesan and I. Iliadis, A general reliability model for data storage systems, In *International Conference on Quantitative Evaluation of Systems (QEST)*, pages 209-

219, 2012.

[107] I. Iliadis, R. Haas, X. Hu and E. Eleftheriou, Disk Scrubbing Versus Intra-disk Redundancy for RAID Storage Systems, *ACM Transactions on Storage*, Vol. 7, No. 2, 2011.

[108] I. Iliadis and V. Venkastesan, Rebuttal to "Beyond MTTDL: A Closed-Form RAID 6 Reliability Equation", *ACM Transactions on Storage*, Vol. 11, No. 2, 2015.

[109] F. Machida, J. Xiang, K. Tadano, Y. Maeno and T. Horikawa, Performability analysis of RAID10 versus RAID6, In *IEEE/IFIP International Conference on Dependable Computing and Networks (DSN) Fast Abstract*, 2013.

[110] M. D. Beaudry, Performance-related reliability measures for computing systems. *IEEE Transactions on Computers*, Vol. C-27, pages 540-547, 1978.

[111] H. Sun, T. Tyan, S. Johnson, R. Elling, N. Talagala, and R. B. Wood, Performability Analysis of Storage Systems in Practice: Methodology and Tools, In *International Conference on Service Availability*, pages 62-75, 2006.

[112] A. Thomasian, and M. Blaum, Mirrored disk organization reliability analysis, *IEEE Transactions on Computers*, Vol. 55 No.12, pages 1640-1644, 2006.

[113] A. Thomasian, and J. Xu, Reliability and performance of mirrored disk organizations, *Computer Journal*, Vol. 51, No.6, pages 615-629, 2008.

[114] A. Thomasian, and Y. Tang, Performance, reliability, and performability of a hybrid RAID array and a comparison with traditional RAID1 arrays, *Cluster Computing*, Vol. 15, No. 3, pages 239-253, 2012.

[115] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives, In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 289-300, 2007.

[116] D. C. Sawyer, Dependability analysis of parallel systems using a simulation-based approach, NASA-CR-195762, 1994.

[117] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hosopdor, and S. Ng, Disk scrubbing in large archival storage systems. In *IEEE/ACM International Symposium*

on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pages 409-418, 2004.

[118] A. Dholakia, E. Eleftheriou, X.-Y. Hu, I. Iliadis, J. Menon, and K. K. Rao, A new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors, *ACM Transactions on Storage*, Vol. 4, No. 1, pages 1-42, 2008.

[119] N. Mi, A. Riska, E. Smirni, and E. Riedel, Enhancing data availability in disk drives through background activities, In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 492-501, 2008.

[120] B. Schroeder, S. Damouras, and P. Gill, Understanding latent sector errors and how to protect against them, In *USENIX Conference on File and Storage Technologies (FAST)*, pages 71-84, 2010.

[121] X. Wu, J. Li, and H. Kameda, Reliability Modeling of Declustered-Parity RAID Considering Uncorrectable Bit Errors, *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. 80, No. 8, pages 1508-1515, 1997.

[122] E. W. D. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K. K. Rao, P. Zhou, Evaluating the Impact of Undetected Disk Errors in RAID Systems, In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 83-92, 2009.

[123] K. V. Vishwanath and N. Nagappan, Characterizing cloud computing hardware reliability, In *ACM Symposium on Cloud computing (SoCC)*, pages 193-204, 2010.

[124] V. Venkastesan and I. Iliadis, Effect of codeword placement on the reliability of erasure coded data storage systems, In *International Conference on Quantitative Evaluation of Systems (QEST)*, pages 241-257, 2012.

[125] R. Xia, F. Machida, and K. S. Trivedi, A Markov decision process approach for optimal data backup scheduling, In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* workshop, pages 660-665, 2014.

[126] R. Xia, F. Machida, and K. S. Trivedi, A scalable optimization framework for storage backup operations using Markov decision processes, In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 169-178, 2015.

[127] S. Gnanasundaram, and A. Shrivastava, Information storage and management, *Wiley Publishing, Inc.*, 2009.

[128] D. M. Smith, The cost of lost data, *Journal of Contemporary Business Practice* 6, No. 3, 2003.

[129] G. Sun, Y. Dong, D. Chen, and J. Wei, Data backup and recovery based on data de-duplication, In *IEEE International Conference on Artificial Intelligence and Computational Intelligence (AICI)*, vol. 2, pages 379-382, 2010.

[130] J. Kaiser, D. Meister, A. Brinkmann, and S. Effert, Design of an exact data de-duplication cluster, In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1-12, 2012.

[131] D. N. Tran, F. Chiang, and J. Li, Friendstore: cooperative online backup using trusted nodes, In *ACM Workshop on Social Network Systems*, pages 37-42, 2008.

[132] L. Toka, D. Matteo, and P. Michiardi, Online data backup: A peer-assisted approach, In *International Conference on Peer-to-Peer Computing (P2P)*, pages 1-10, 2010.

[133] L. M. Kaufman, Data security in the world of cloud computing, *IEEE Security & Privacy*, No. 4, pages 61-64, 2009.

[134] L. Courtes, O. Hamouda, M. Kaaniche, M-O. Killijian, and D. Powell, Dependability evaluation of cooperative backup strategies for mobile devices, In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 139-146, 2007.

[135] K. Keeton, and A. Merchant, A framework for evaluating storage system dependability, In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 877-886, 2004.

[136] R. Xia, X. Yin, J. Alonso, F. Machida, and K. S. Trivedi, Performance and availability modeling of IT systems with data backup and restore, *IEEE Transactions on Dependable and Secure Computing*, Vol. 11, No. 4, pages 375-389, 2014.

[137] K. Renuga, S. S. Tan, Y. Zhu, T. Low, and Y. Wang, Balanced and efficient data

placement and replication strategy for distributed backup storage systems, In *International Conference on Computational Science and Engineering (CSE)*, Vol. 1, pages 87-94, 2009.

[138] S. Gaonkar, K. Keeton, A. Merchant, and W. H. Sanders, Designing dependable storage solutions for shared application environments, *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pages 366-380, 2010.

[139] Z. Sanaei, S. Abolfazli, A. Gani, and R. Buyya, Heterogeneity in mobile cloud computing: Taxonomy and open challenges, *IEEE Communications Surveys & Tutorials*, No. 99, pages 1-24, 2013.

[140] F. Machida and N. Miyoshi, Effective server procurement for mobile thin-client service by simulating virtual machine reallocation, Book chapter in *Stochastic Operations Research in Business and Industry*, World Scientific, Singapore, accepted.

[141] F. Machida, S. Kohno, S. Maebara, M. Nakagawa, Just-in-time server procurement to private cloud for mobile thin-client service, In *International Conference on Network and Service Management (CNSM)*, 2015.

[142] T. Zheng, M. Woodside, M. Litoiu, Performance model estimation and tracking using optimal filters, *IEEE Transactions on Software Engineering*, Vol. 34. No. 3, pages 391-406, 2008.

[143] P. A. Dinda, D. R. O'Hallaron, Host load prediction using linear models, *Cluster Computing*, Vol. 3. Issue. 4, pages 265-280, 2000.

[144] Y. Zhang, S. Wei, and Y. Inoguchi, CPU load predictions on the computational grid, *IEICE Trans. Inf. & Syst.*, Vol. E90-D, No. 1, 2007.

[145] D. M. Lucantoni, K. S. Meier-Hellstern, and M. F. Neuts, A single server queue with server vacations and a class of non-renewal arrival processes, Advances in Applied Probabilities, vol. 22, pages 676–705, 1990.

[146] G. Horváth, P. Buchholz, and M. Telek, A MAP fitting approach with independent approximation of the inter-arrival time distribution and the lag correlation, In *International Conference on the Quantitative Evaluation of Systems (QEST),* pages 124–133, 2005.

[147] H. Okamura, and T. Dohi, mapfit: An R-based tool for PH/MAP parameter estimation, In *International Conference on the Quantitative Evaluation of Systems (QEST),* pages 105-112, 2015.

[148] B. Korte, J. Vygen, Combinatorial Optimization: Theory and Algorithms, Japanese Edition, *Springer*, 2005.

[149] N. Bobroff, A. Kochut, and K. Beaty, Dynamic placement of virtual machines for managing SLA violations, In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 119-128, 2007.

[150] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, Black-box and gray-box strategies for virtual machine migration, In ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2007.

[151] F. Hermenier, X. Lorca, J. M. Menaud, G. Muller, and J. Lawall, Entropy: a consolidation manager for clusters. In International Conference on Virtual Execution Environments (VEE), pages 41-50, 2009

[152] Y. Ajiro, Recombining Virtual machines to autonomically adapt to load changes, In Annual Conference of the Japanese Society for Artificial Intelligence, (In Japanese) 2008.

[153] L. Deboosere, B. Vankeirsbilck, P. Simoens, F. De Turck, B. Dhoedt, and P. Demeester, Cloud-based desktop services for thin clients, *IEEE Internet Computing*, Vol. 16, No. 6, pages 60-67, 2012.

[154] Y. Kirihata, Y. Samashima, and T. Onoyama, WriteShield: A pseudo thin-client system for prevention of information leakage, In *IEEE International Conference on Industrial Informatics (INDIN)*, pages 478-483, 2010.

[155] P. Calyam, R. Patali, A. Berryman, A. Lai, and R. Ramnath. Utility-directed resource allocation in virtual desktop clouds, *Computer Networks*, Vo. 55, No. 18, pages 4112-4130, 2011.

[156] D. Schlosser, B. Staehle, A. Binzenhofer, and B. Boder, Improving the QoE of Citrix thin client users, In *International Conference on Communications (ICC)*, pages 1-6, 2010.

[157] A. Kochut, K. Beaty, H. Shaikh, D. G. Shea, Desktop workload study with implications for desktop cloud resource optimization, In *IEEE International Symposium on Parallel and Distributed Processing Workshops*, pages 1-8, 2010.

[158] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, Workload analysis and demand prediction of enterprise data center applications, In *IEEE International Symposium on Workload Characterization*, pages 171-180, 2007.

[159] M. Platania, D. Obenshain, T. Tantillo, R. Sharma, and Y. Amir, Towards a practical survivable intrusion tolerant replication system, In *IEEE International Symposium on Reliable and Distributed Systems (SRDS)*, pp. 242-252, 2014.

[160] F. Machida, J. Xiang, K. Tadano, and S. Hosono, An asset-based development approach for availability and safety analysis on a flood alert system, In *International Workshop on Recent Advances in the DependabIlity AssessmeNt of Complex systEms (RADIANCE)*, pages 51-56, 2015.

[161] J. Dunjo, V. Fthenakis, J. Vilchez and J. Arnaldos, Hazard and Operability Analysis: A literature review, *Hazardous Materials*, Vol. 173, No 1-3, pages 19-32, 2010.

[162] N. G. Leveson, Engineering a safer world: Systems Thinking Applied to Safety, *MIT Press*, 2012.

[163] Business Insider, The massive AWS outage hurt 54 of the top 100 internet retailers — but not Amazon, accessed in December 2017. http://www.businessinsider.com/aws-outage-hurt-internet-retailers-except-amazon-2017-3

[164] Web Host Industry Review, Microsoft Azure storage issues caused by two incidents, accessed in December 2017. http://www.thewhir.com/web-hosting-news/microsoft-azure-storage-issues-caused-by-two-incidents