1-1-2007

Effects of software aging and rejuvenation on performability of layered distributed systems

Jigar Patel
Ryerson University

EFFECTS OF SOFTWARE AGING AND REJUVENATION ON PERFORMABILITY OF LAYERED DISTRIBUTED SYSTEMS

by

Jigar Patel, B.Eng.
S.P.University, India, 2005

A thesis presented to Ryerson University
in partial fulfillment of the
requirements for the degree of
Master of Applied Science
in the program of
Electrical and Computer Engineering

Toronto, Ontario, Canada, 2007

EFFECTS OF SOFTWARE AGING AND REJUVENATION ON PERFORMABILITY OF LAYERED DISTRIBUTED SYSTEMS

© Jigar Patel

Master of Applied Science

Department of Electrical and Computer Engineering

Ryerson University, 2007

Abstract

When a fault-tolerant layered distributed system continues its operation despite the presence of component failures, its performance is usually degraded. Its performance can also be degraded if it is executing continuously for a long period of time due to a phenomenon known as software aging. To prevent unexpected or unplanned outages due to aging, a pro-active technique called software rejuvenation can be employed. This technique involves gracefully terminating an application and immediately restarting it with a refreshed internal state. For proper modeling of these systems, their performance and dependability characteristics need to be considered in a unified way, called performability. This thesis proposes a new model called "Rejuvenated-FTLQN", to evaluate the effects of software aging and rejuvenation on performability of these layered systems. Specifically a Layered Queueing Network (LQN) is used for performance analysis and a Multi State Fault Tree (MSFT) is used for dependability analysis. The model is also used to study the impact of performing rejuvenation, time to perform rejuvenation and rejuvenation frequency on performability of a system. A software tool called "Rejuvenated-FTLQNS" has been developed to automate the model solution.

Acknowledgements

First, I would like to thank my supervisor, Prof. Olivia Das for her unequivocal support and guidance over a long period of time. I am very fortunate to have worked with her during the entire course of my graduate studies. I will never forget the countless moments, that we have spent discussing new ideas. Her advice, encouragement, kindness and patience have been a source of great inspiration to me.

Special thanks are due to Prof. Alagan Anpalagan and Prof. Olivia Das for working together to ensure the series of scholarships I have received. I would also like to thank the Government of Ontario for awarding me Ontario Graduate Scholarship (OGS) and Natural Sciences and Engineering Research Council (NSERC) for funding the research.

Words cannot express my gratitude towards my father (Naresh), mother (Kirti) and sister (Khushbu) for all their support and unconditional love. Hellen Keller has truly said *"The best and most beautiful things in the world cannot be seen or even touched - they must be felt with the heart"*. I dedicate this thesis to my mother.

Table of Contents

List of Tables

List of Figures

List of Appendices

List of Acronyms

CTMC - Continuous Time Markov Chain

DTMC - Discrete Time Markov Chain

FTLQN - Fault Tolerant Layered Queueing Network

FTLQNS - Fault Tolerant Layered Queueing Network Solver

LQN - Layered Queueing Network

LQNS - Layered Queueing Network Solver

MSFT - Multi State Fault Tree

OLTP - On-Line Transaction Processing

PERL - Practical Extraction and Report Language

QN - Queueing Network

SHARPE - Symbolic Hierarchical Automated Reliability and Performance Evaluator

SOAP – Simple Object Access Protocol

SSP - Steady State Performability

Chapter 1
Introduction

1.1 Introduction and Motivation

In recent years, our dependency on complex distributed systems for carrying out chores of our daily life has increased dramatically. For example, banking, health care, data communication, telecommunication tasks inevitably involves interaction with these systems. Since outages of these systems incur high cost and may result in human loss, there has been widespread research on designing these systems to be able to tolerate failures. Research has repeatedly found that the main cause of system outage is due to software failures [28]. Software faults are classified by Gray [28] into Bohrbugs and Heisenbugs. Bohrbugs are essentially permanent design faults and hence almost deterministic in nature. They can be identified easily and weeded out during the testing and debugging phase (or early deployment phase) of the software life cycle. Heisenbugs, on the other hand, are essentially permanent faults whose conditions of activation occur rarely or are not easily reproducible. Trivedi et al. [29] introduced another kind of software fault called aging-related fault where once the software is started, potential fault conditions gradually accumulates with time leading to either performance degradation or transient failures or both. This process of software degradation while executing continuously for a long period of time is known as **Software Aging.**

Software aging is observed in Internet explorer, Netscape, as well as commercial operating systems. The possible symptoms of software aging are memory leaking, unreleased file locks, file descriptor leaking, data corruption in the operating environment of system resources etc. Thus it is caused by the bugs in the application program that was developed, in the libraries that the application is using or in the application execution environment (e.g. operating system) [31]. Software aging has been observed in

telecommunication systems [31], OLTP (On-Line Transaction Processing) systems [7], web-servers [36], SOAP-based server [48], safety-critical systems [37], space-craft systems [52]. The most glaring example of software aging that resulted in loss of human life is US Patriot missiles deployed during Gulf war [37]. The accumulated round off error led to the interpretation of an incoming Iraqi Scud missile as a false alarm which cost the lives of 28 US soldiers.

To counteract this phenomenon of software aging, a pro-active approach to fault management called software rejuvenation aimed to prevent unexpected or unplanned outages due to aging may be used. **Software rejuvenation** is the concept of gracefully terminating an application and immediately restarting it with a refreshed internal state [31]. Cleaning the internal state of software might involve garbage collection, flushing operating system kernel tables, reinitializing internal data structures, and hardware reboot. Software rejuvenation has been successfully implemented in various systems like IBM xSeries servers [32], scientific speech synthesis system [31] and web-servers [66]. The failure which resulted in loss of human lives (Patriot missiles) could have been prevented if the computer was restarted after each 8 hours of running time [37].

Enterprises must continuously provide high quality of service (QoS) to gain a competitive advantage and in order to meet the ever growing demands for increasing number of users and additional services. The need for integrating evaluation of QoS requirements like performance and dependability, into the software development process has been recognized long time ago. For new applications where measurements are not possible or even for existing applications, **analytical modeling** can be used for predicting the behaviour of the system. Simulations models can be build to any arbitrary level of detail and can provide very good estimates about the dynamic behaviour of the system. However the time and resource cost, to build and analyze simulation models can be prohibitive. Analytic models, on the other hand, though may not always encompass the full dynamics of the system, provide a good balance between cost and accuracy. Analytic models are used to gain deeper insight into the system and to help choose among alternative designs, configurations, and so forth by quickly answering 'what-if' questions.

Because analytic models can be build and solved quickly, they make ideal candidates for such purpose.

Performance and dependability are the two most important quality requirements of a system. Pure performance modeling involves representing the probabilistic nature of user demands and predicting the system capacity to perform, under the assumption that the system structure remains constant [68]. Pure dependability modeling deals with the representation of changes in the structure of the system, generally due to faults, being modeled. For modeling fault-tolerant systems that are capable to provide continued service in presence of failures (may be in a degraded mode), their performance and dependability should be considered simultaneously, known as **performability** [38]. Performability modelling considers the effect of structural changes in response to failures and their impact on the overall performance of the system.

Business and industry is moving to the "client-server" computing paradigm. With this computing model, clients with varying degrees of sophistication are connected to one or more servers. The servers run applications on behalf of the clients or store some resource such as data or perform both functions. In such distributed systems, processing of request from users is distributed among several tasks. Most of the distributed systems used today are structured in layers with some kind of user interface task in the topmost layer making requests to different layers of servers [60]. Software aging can occur at any layer of layered distributed system and rejuvenation may be implemented to combat its effects. Thus the evaluation of performability of such layered systems, in presence of aging and rejuvenation is needed.

1.2 Problem Definition and Research Objectives

Nowadays, most of the distributed systems with clients and servers are typically constructed with layered software architecture. The Layered Queueing Network (LQN) model can be used to study the performance of such layered systems. LQN model is an extension of widely used Queueing Network (QN) model. In fault tolerant layered

distributed systems like the telecommunication systems and banking systems, the effects of a server failure are felt through the inability of its clients to obtain service. Thus failures are propagated by the layered dependencies. In [9], a model, known as Fault Tolerant Layered Queueing Network Model (FTLQN), was developed to express these layered service failure and repair dependencies and an algorithm was provided for computing performability measures. The FTLQN model has two advantages, (1) it closely resembles the software architecture (making it easy to build), and (2) it contains the service dependencies which are required in one form or another to analyze the failures. However, FTLQN model does not consider the effects of aging and rejuvenation for software components. It would be valuable to include these effects to get a better understanding of the overall behaviour of the system. Figure 1.1 shows the main concepts involved in the problem definition.

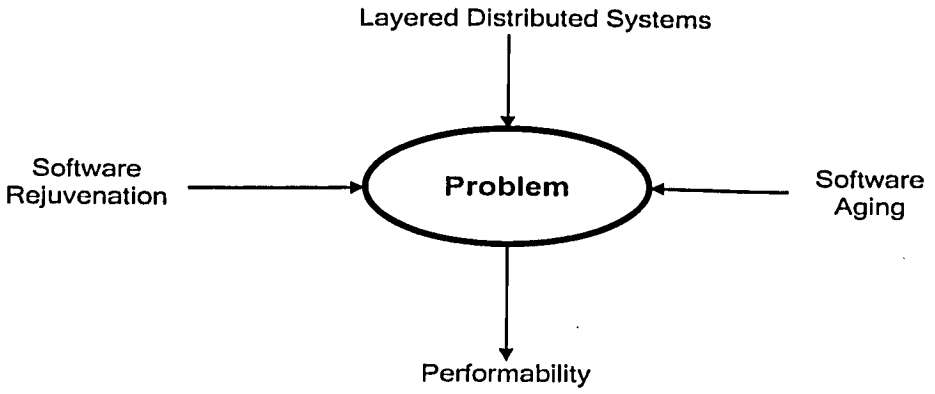

Figure 1.1 - Main Ingredients of the Problem

The main goal of this thesis is to incorporate the effects of software aging and rejuvenation into FTLQN performability evaluation. So the question raised is: How the phenomenon of software aging and rejuvenation on the servers is affecting the performability of the system? The new model termed as "**Rejuvenated-FTLQN**" is introduced to answer this question. It considers the propagation of performance degradation due to aging from lower to higher layers of servers. The performance degradation due to aging is modeled by changing the service rate of the software tasks.

In this work, Multi-State Fault Trees (MSFTs) are used for dependability analysis and Layered Queueing Networks (LQNs) are used for performance analysis. MSFTs are solved using the SHARPE (*Symbolic Hierarchical Automated Reliability and Performance Evaluator*) tool [53] and the LQNs are solved using the LQNS (*Layered Queueing Network Solver*) tool [21]. The model solution avoids solving very large Markovian models. A software tool called Rejuvenated-Fault Tolerant Layered Queueing Network Solver (**Rejuvenated-FTLQNS**) has been developed using C++ and PERL to automate the model solution technique as described in chapter 3 of this thesis.

1.3 Why this Problem is Important?

The problem of computing performability of fault tolerant layered distributed systems under the effects of software aging and rejuvenation is important because:

- Server processes at different layers are intended to run continuously forever except during software upgrades and they may start aging after a certain period of time providing degraded service. This aging phenomenon affects the performance of the system. Also, due to aging the application exhibits increasing failure rate.

- Rejuvenation may be implemented to counteract the effects of software aging, and the system may be unavailable while undergoing rejuvenation affecting the dependability of the system.

Thus taking aging and rejuvenation into account while computing performability, will give more realistic measure of overall quality of responsiveness compared to when it is not considered.

1.4 Comparison with Related Previous work

In this work, prediction-based rejuvenation policy [57] [58] is assumed as opposed to time-based rejuvenation. In prediction-based rejuvenation policy, the rejuvenation starts whenever a degraded state of the component is detected by means of analyzing some observable symptoms. Otherwise, the component eventually goes to an undetected failed state that usually requires higher detection and repair time than the rejuvenation time. In time-based rejuvenation policy, rejuvenation is done after particular time interval. Time-based rejuvenation is widely used in some real production systems E.g. web-servers [66].

A survey about papers that follow the analytic-based approach for prediction based rejuvenation policy can be found in [57]. A semi-markov model that relaxed the assumption for exponential distribution was presented in [19]. A markov regenerative process that allowed the rejuvenation trigger clock to start in a robust state was described in [27]. A measurement based approach for proactive detection of software aging in OLTP servers was studied in [7] using monitored data collected during a period of 5 months. That data was used to train a pattern-recognition tool. After the training phase, the system was used to monitor the production environment. That tool was able to predict the occurrence of software aging with a long time in advance.

The model we are using to capture the effects of software aging and rejuvenation is similar to [31] [57] [58] in that it uses hypoexponentially distributed (which is an increasing failure rate distribution) time to failure for modelling aging. This two-stage failure process is confirmed by performing practical experiment as described in [33]. In this experiment 20 workstations were connected to the server and it was observed that in the operational phase, the server experiences performance degradation, characterized by lower and lower internet access speed. The data from the server system was periodically monitored and collected. The two quantities measured were (1) response time, the interval from the time a client sends out the first byte of request until it receives the first

byte of reply and (2) the available physical memory. The response time had an increasing trend and available memory had a decreasing trend as described in [33].

The two-stage failure process was also used by Rinsaka and Dohi, for behavioral analysis of single-version and two-version software system [46]. Garg et al. [25] considered the effect of load on aging. Decreasing service rate was used to capture degradation caused by software aging in [44]. However, most of the above mentioned work on aging and rejuvenation was focused on evaluation of availability aspect of the system. For e.g. in [31], steady state availability was measured and in [27] the number of jobs lost was evaluated. In this work, performability of the system is evaluated. Our work differs from previous works in following aspect: The application domain for Rejuvenated-FTLQN model is layered distributed systems or multi-layered system, where the failure of a service depends on other services in lower layers. Thus the aging in layered system with multiple layers of queues is modeled. None of the previous works has done so. The main strength of our model is its capability of capturing the aging in a system with layers of queues.

Figure 1.2 compares the Rejuvenated-FTLQN model with two of the existing models namely LQN and FTLQN. LQN [60] is a pure performance model. In the layered view, there are server tasks which have their own queues of messages to serve and which in turn make requests to lower layer servers. A Layered Queueing Network (LQN) model is a performance modelling framework for client-server like distributed systems that use a style of synchronous inter-task communication.

The FTLQN [9] performability modelling concept can be applied to model a class of systems which possess a fault-tolerant client-server like structure and whose design takes advantage of the stand-by redundancy approach, an approach that advocates the usage of a stand-by server when the primary server fails and a mechanism to re-direct the service requests at the time of failure. Since this is non-state space based model, it does not suffer from state space explosion problem. FTLQN model considers layered distributed systems and Rejuvenated-FTLQN model is an extension of FTLQN model. The choice of

selecting FTLQN model over other performability model (e.g. markov reward model) is because of two main reasons: (1) The application domain is layered systems (2) It avoids state space explosion problem. Rejuvenated-FTLQN model considers the effects of software aging and rejuvenation on the software components. Performance degradation and increasing failure rate caused by software aging and change in unavailability of the system caused by software rejuvenation is considered while evaluating performability.

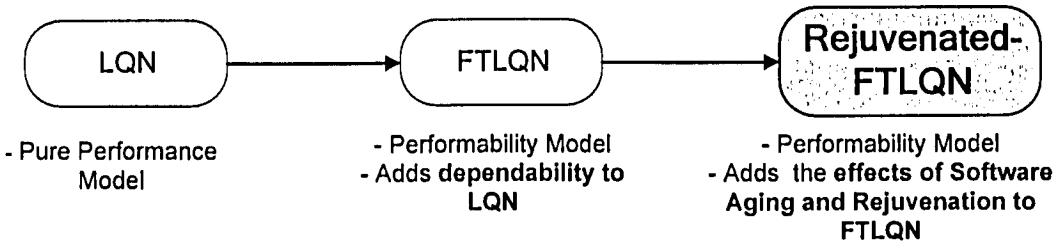

Figure 1.2 – Comparison of LQN, FTLQN and Rejuvenated-FTLQN

1.5 Contributions

The principal contributions of this thesis are as follows:

- A new model called "Rejuvenated-FTLQN" has been developed for computing performability of layered fault tolerant client-server like distributed systems under the effects of software aging and rejuvenation on servers. The primary contribution being the use of MSFT (Multi State Fault Tree) in the model solution technique (chapter 3). The model solution can be briefly outlined as below:

    - *Modeling Aging and Rejuvenation*: A Continuous Time Markov Chain (CTMC) is added for every software task (process) to the FTLQN model to take into account the effects of software aging and rejuvenation. CTMC denotes the state of a software task and transitions between states.
    - *Operational Configurations*: Fault-tolerance and software aging gives rise to the different configurations of the system in which the system is

operational. Determine all the different operational configurations of the system using the AND-OR graph representation of the FTLQN model.

- *Performance Analysis*: Evaluate the Layered Queuing Network (LQN) model for every configuration to obtain the performance measure that will be assigned to every operational configuration.
- *Dependability Analysis*: A Multi State Fault Tree (MSFT) is used to calculate the probability of the system being in each of the operational configuration.
- *Steady State Performability calculation*: Combine the results from the performance analysis and the dependability analysis to obtain the performability measure.

- A software tool termed as *Rejuvenated-FTLQNS* (Fault-Tolerant Layered Queueing Network Solver) has been developed to automate the Rejuvenated- FTLQN model solution (chapter 5). This tool is used to analyze the effects of software aging and rejuvenation on performability of large Building Security System (chapter 6).

- The impact of (1) rejuvenation frequency, (2) time to perform rejuvenation and (3) the interval in which the application goes from healthy state to failure probable state due to aging on performability is analyzed (chapter 4).

1.6 Outline of the Thesis

Chapter 2 provides background information that forms the foundation of our work. The main concepts discussed are performability, software aging, rejuvenation, layered distributed systems and FTLQN.

Chapter 3 presents a detailed description of the Rejuvenated-FTLQN model which is developed in present research and the model solution technique for computing performability with the help of an example.

Chapter 4 investigates the impact of rejuvenation on the system by comparing the model solution in two cases - with and without rejuvenation. This chapter also discusses the

effects of rejuvenation frequency, time to perform rejuvenation and base longevity interval on performability.

Chapter 5 presents the high level description of the software tool (Rejuvenated-FTLQNS) that automates the model solution technique.

Chapter 6 deals with the application of Rejuvenated-FTLQN model on the large Building Security System (BSS).

Chapter 7 concludes the study, summarizes the work and provides some directions for future scope of studies.

Chapter 2
Background

This chapter introduces the preliminary concepts that form the foundation of our work. These concepts are used throughout this thesis. In section 2.1, three main techniques used for evaluation of a system are described along with their pros and cons. Analytical modeling is the technique used in the present research. Section 2.2 deals with three different kinds of software faults, including the aging-related faults. Section 2.3 describes the phenomenon of software aging and rejuvenation along with some practical examples. Section 2.4 discusses about various quality attributes of the system and defines the term performability. Section 2.5 describes fault tree and multi state fault tree with the help of an example and section 2.6 describes markov chain. In section 2.7 an overview of layered distributed systems is given. Section 2.8 deals with the Layered Queuing Network (LQN), which is a performance model for layered distributed systems. An extension of LQN model called Fault Tolerant Layered Queuing Network (FTLQN) is discussed in section 2.9. Different modeling approaches for software rejuvenation namely analytical modeling and measurement-based approach are discussed in section 2.10, with more emphasis on analytical approach. Finally, various performability measures including steady state performability and different performability models are reviewed in section 2.11.

2.1 Analytical Modeling

There are three main techniques for understanding the dynamic behavior of a computer system [35]. When the system under study already exists and is accessible, we can make **measurements** by performing experiments. When the system does not exist or it is clumsy to deal with, a model must be developed. **Analytical models** use mathematical concepts and notation. **Simulation** models are computer programs that mimic the behaviour of the system, under some assumptions. Both kind of models restricts to the important aspects only and leave out other details.

While measurement is an attractive option for assessing an existing system or a prototype, it is not a feasible option during the system design and implementation phases. Model-based evaluation has proven to be an attractive alternative in these cases. Several types of models are currently used in practice. The most appropriate type of model depends upon the complexity of the system, the questions to be studied, the accuracy required, and the resources available for the study.

Discrete-event simulation is also another widely used modeling technique in practice but it tends to be relatively expensive. Analytical modeling provides a cost-effective alternative to simulation for studying the performance and dependability of computer and communication systems. Due to recent developments in model generation, solution techniques and automated tools, large and realistic models could be developed and studied.

A model is an abstraction of a system: an attempt to distill, from the mass of details that is the system itself, exactly those aspects that are essential to the system's behavior. Once a model has been defined through this abstraction process, it can be parameterized to reflect any of the alternatives under study and then evaluated to determine its behavior under this alternative. Using a model to investigate system behavior is less laborious and more flexible than experimentation, because the model is an abstraction that avoids unnecessary detail. It is more reliable than intuition, because it is more methodical: each

particular approach to modelling provides a framework for the definition, parameterization, and evaluation of models. Of equal importance is that using a model enhances both intuition and experimentation.

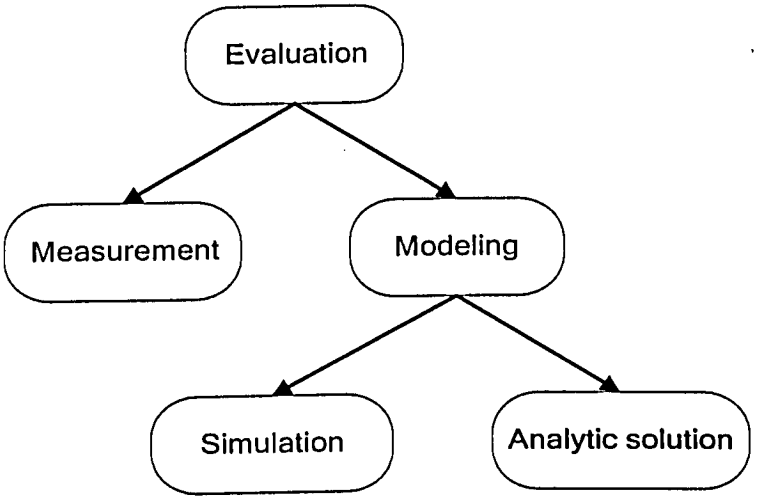

Figure 2.1 - Techniques for evaluation of the system

The choice of the technique depends on the type of the system investigated, its availability, familiarity with the techniques, time and resource constraints, desired accuracy etc. The advantages of analytical modeling are that the time required is generally less than other two techniques and even the cost is lower than performing simulations and measurements through experiments. The trade-off evaluation is even easier in case of analytical modeling, and it can be used at any stage of the system life cycle.

There are two main reasons for modeling a given system:

- Existing systems are modeled for better understanding, for analyses of deficiencies such as identification of potential bottlenecks, or for upgrading studies.
- Models are used during the design of future systems in order to check whether requirements are met.

Different levels of details are required for both the above mentioned cases. A high level description of the model is the first step to be accomplished. Either information about a real computer system is used to build the model, or experiences gained in earlier modeling studies are implicitly used. This process is rather complicated and needs both modeling and system application-specific expertise. Models always have a specific purpose for which it is build, and which determines its structure and representation. Deciding which technique to use is often based on the following criteria [21]:

Stage: point in life cycle when study is to take place.
Time required: when the results are needed.
Tools: analytic modelling tools, simulators, measurement packages.
Accuracy: degree to which results match reality.
Trade-off evaluation: ability to study different system configurations.
Cost: time and money needed to conduct the study.

For the solution to the problem defined in chapter 1, analytical modeling technique is used.

2.2 Classification of Software Faults

This section describes the various types of software faults. Gray [28] classifies software faults into ***Bohrbugs*** and ***Heisenbugs***.

- **Bohrbugs** are essentially permanent design faults and hence almost deterministic in nature. They can be identified easily and weeded out during the testing and debugging phase (or early deployment phase) of the software life cycle.

- **Heisenbugs**, on the other hand, are design faults that behave in a way similar to hardware transient or intermittent faults. Their conditions of activation occur rarely or are not easily reproducible. These faults are extremely dependent on

the operating environment (other programs, OS and hardware resources). Hence these faults result in transient failures, i.e., failures which may not recur if the software is restarted. Some typical situations in which Heisenbugs might surface are boundaries between various software components, improper or insufficient exception handling and interdependent timing of various events. It is for this reason that Heisenbugs are extremely difficult to identify through testing. In fact, any attempt to detect such a bug may alter the operating environment enough to change the symptoms. A mature piece of software in the operational phase, released after its development and testing stage, is more likely to experience failures caused by Heisenbugs than due to Bohrbugs.

Trivedi et al [29] designates faults attributed to software aging, which are quite different from bohrbugs and heisenbugs, as aging-related faults.

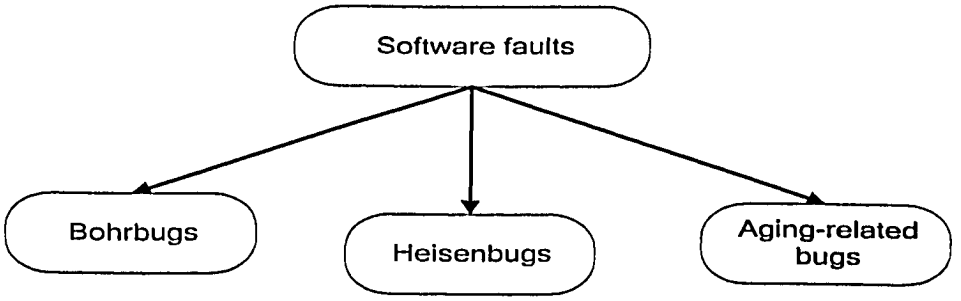

Figure 2.2 - Classification of Software faults

- **Aging-related bugs** are the faults that cause deterioration of the operating system resources, data corruption and numerical error accumulation. Examples include memory leaks, unreleased file locks, storage space fragmentation, accumulation of round off errors etc. The fault conditions gradually accumulate over time and lead to performance degradation of the software or transient failures or both.

2.3 Software Aging and Rejuvenation

The phenomenon of *software aging* has been reported by several recent studies [23] [31] [33]. It was observed that once the software was started, potential fault conditions gradually accumulated with time leading to either performance degradation or transient failures or both. Failures may be of crash/hang type or those resulting from data inconsistency because of aging. Typical causes of aging are:

- Memory bloating or leaking,
- Unreleased file-locks,
- Data corruption,
- Storage space fragmentation and
- Accumulation of round off errors.

According to Parnas [43] there are two distinct types of software aging. First is caused by the failure of the product's owners to modify it to meet changing needs and second is the result of the changes that are made. Unless software is frequently updated, its user will become dissatisfied and they will switch towards new product as soon as the benefits overweigh the costs of retraining and converting. The software will be referred as old and outdated. If the program is large, understanding the original design and finding those sections or modules that must be changed is challenging task. Changes made by people without understanding the original design almost always cause the structure of the program to degrade. Sometimes the damage is small but often it is quite severe. After many such changes, the original design rules are even violated in some cases.

Thus we have two different views on software aging. One refers to the need of modifying the software due to changing requirements as well as change in surrounding environment as stated by Parnas [43]. While other refers to performance degradation of software due to factors like memory bloating and leaking, unreleased file-locks, data corruption, storage space fragmentation and accumulation of round-off errors as stated by Kishor Trivedi et al. [56].

To counteract first type of software aging one should design software for change; and the documentation must be well organized, complete and precise. To apply information hiding, one must begin by characterizing the changes that are likely to occur over the lifetime of the product. Even if we take all reasonable preventive measure, aging is inevitable.

To counteract second type software aging as described in [56] a technique called software rejuvenation is implemented. Software rejuvenation is a proactive approach of fault management which involves gracefully terminating an application or a system and restarting it in a clean internal state [31]. For analysis and modeling purpose in this thesis, this type of software aging is considered. From now onwards aging refers to this software aging unless stated otherwise.

Software rejuvenation involves halting the running software occasionally, "cleaning" its internal state and restarting it. Some examples of cleaning the internal state of software might involve:

- Garbage collection.
- Flushing operating system kernel tables.
- Reinitializing internal data structures.
- An extreme, but well known example of rejuvenation is a hardware reboot.

Practical Examples of Software Aging and Rejuvenation

1. US Patriot Missiles

The software fault in the Patriot missile-defense system responsible for the Scud incident in Dhahran was due to aging-related bug [37]. To project a target's trajectory, the weapons-control computer required its velocity and the time as real values. However, the system kept time internally as an integer, counting tenths of seconds and storing them in a

24-bit register. The necessary conversion into a real value caused imprecision's in the calculated range where a detected target was expected next. For a given velocity of the target, these inaccuracies were proportional to the length of time that the system had been continuously running. As a consequence, the risk of failing to track, classify, and intercept an incoming Scud missile increased with the length of time that the Patriot missile-defense system operated without a reboot.

On 21 February 1991, the Patriot Project Office warned Patriot users that "very long runtimes" could negatively affect the system's targeting, implying it should be rebooted regularly. Unfortunately, the army officials assumed that the users would not continuously operate the Patriot systems long enough for a failure to become imminent; therefore, they did not specify the required rejuvenation frequency.

2. Apache Web server

Apache is the one of most popular web server software currently used [65]. Apache provides some software rejuvenation features. For example, a child process is terminated and restarted after handing more than a specified number of requests. Apache also allows system administrators stop and restart the web server software in three different ways by sending different signals to the parent process:

i. TERM signal (stop now)
ii. HUP signal (restart now)
iii. USR1 signal (graceful restart)

3. Log-File System

The log-file system endures performance degradation when the disk space is fragmented, a phenomenon of software aging [40]. A log-structured file system writes all new information to disk in a sequential structure, name the log. The fundamental purpose of a log-structured system is to improve write-performance by buffering a sequence of file system changes in the file cache and then writing all the changes to disk, sequentially, in a single disk write operation. In order to maintain a large free area on the disk for fast

writing, the log is divided into segments, and uses a segment cleaner to compress the live information from heavily fragmented segments. Segment cleaning can be considered as a rejuvenation operation action to counteract the performance degradation induced by file system fragmentation.

4. Operating Systems

Operating systems such as Windows and Linux suffers from memory leaks. Memory leaks are caused by software residual bugs, which prevent a program from freeing up the memory that it no longer needs. As a result of memory leaks, the program grabs more and more memory until it finally crashes when there is no memory available.

5. Netscape and Internet Explorer

Client applications like Netscape and Internet Explorer also suffers from memory leaks, which leads to occasional crash/hang of the application.

2.4 Quality Attributes

Computer systems are used in many critical applications where a failure can have serious consequences (loss of lives or property). Developing systematic ways to relate the software quality attributes of a system to the system's architecture provides a sound basis for making objective decisions. The ultimate goal is the ability to quantitatively evaluate and trade off multiple software quality attributes to arrive at a better overall system. Software quality is the degree to which software possesses a desired combination of attributes.

- **Performance** - *"Given that it works, how well it works?"*

The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy or memory usage [35]. Latency or the response time and the throughput are two most important performance parameters.

- **Reliability** – *"For how long it works?"*

Reliability refers to the ability of the system to operate continuously without interruption [54]. *"The ability of an item to perform a required function under given conditions for a given time interval"*. Reliability is defined as the probability that the system functions properly and continuously in the interval [0; t], assuming that it was operational at time 0.

- **Availability** – *"Does it Work?"*

Availability refers to the accessibility of the system to users. A system is available if its users' requests for service are accepted at the time of their submission [54]. Unlike reliability, availability is instantaneous. The former focuses on the duration of time a system is expected to remain in continuous operation or effectively operational. The latter concentrates on the fraction of time instants where the system is operational in the sense of being accessible to the end user. *"The ability of an item to be in a state to perform a required function at a given instant of time or at any instant of time within a given time interval, assuming that the external resources, if required, are provided."*

An important difference between reliability and availability is that reliability refers to failure-free operation during an interval, while availability refers to failure-free operation at a given instant of time.

- **Dependability**

Dependability is that property of a computer system such that reliance can justifiably be placed on the service it delivers [2]. Dependability has several attributes, including:

- ***Availability*:** readiness for usage.
- ***Reliability***: continuity of service.
- ***Safety:*** non-occurrence of catastrophic consequences on the environment.

- *Confidentiality*: non-occurrence of unauthorized disclosure of information.
- *Integrity:* non-occurrence of improper alterations of information.
- *Maintainability*: aptitude to undergo repairs and evolution

- **Performability**

The implicit assumption in the analysis of availability and reliability is that the relevant system states are binary: either the system is up or running or it is not. This simplistic view does hold true for systems that cannot tolerate failures, but for fault-tolerant systems, many more system states become important, one for every possible masked failure pattern. Under such partial failures, the system's performance degrades, even as its full range of functionality remains intact. One way to measure the consequences is to reward the system for every time unit it is ready, at a rate proportional to its performance during that interval. Thus Performability measures probabilistically quantify a systems "ability to perform" in a given operational environment. *Performability* is a composite measure of a system's performance and its dependability. This measure is the vital evaluation method for *degradable* systems - *highly dependable systems which can undergo a graceful degradation of performance in the presence of faults (malfunctions) allowing continued "normal" operation* [38]. A more detailed discussion on performability can be found in the section 2.11.

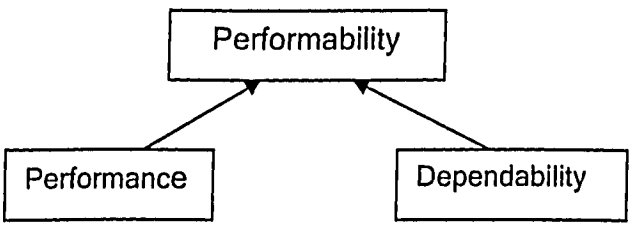

Figure 2.3 - Performability – a composite measure

2.5 Fault Tree and Multi State Fault Tree (MSFT)

A *fault tree* [54] represents the combination of component failures that cause the occurrence of system failure in a tree-like structure. It uses boolean gates (AND, OR and

k-out-of-n) to represent the combinations. If two gates share an input, then the fault tree is said to have repeated events. The fault tree is a pictorial representation of the combination of conditions that can cause the occurrence of system failure. Fault tree is one of the most commonly used models for reliability analysis. A condition at higher level is reduced to a combination of lower level conditions by means of logic gates. The process of reduction stops when the basic conditions are reached wherein a basic condition is a condition that cannot be reduced further. The component failures can be the basic conditions. It is assumed that the basic conditions are mutually independent and that their probabilities are known. The condition is denoted by a value of logic 1 for failure at a node; otherwise the logic value of the node is 0. Each gate has inputs and outputs. The input to a gate is either a basic condition or the output of another gate. The output of an *and* gate is a logic 1 if and only if all of its inputs are logic 1. The output of an *or* gate is a logic 1 if and only if one or more of its inputs are at logic 1. The output of a k out of n gate is logic 1 if k or more of the inputs are at logic 1. There is a single output called the top condition representing the occurrence of system failure.

- **Example – Fault Tree**

Suppose we have a system with two processors having fast private memory modules and the system having slower, shared memory modules. We assume that the system operates as long as there is at least one operational processor with access to either a private or shared memory. The fault tree will be as shown in figure 2.4.

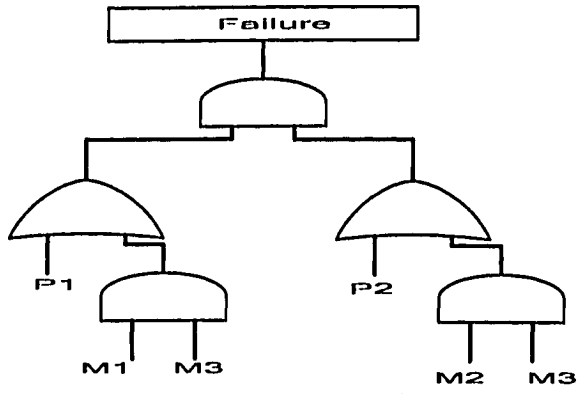

Figure 2.4 - A Fault Tree

Similar to a fault tree, Multi State Fault Tree (MSFT) [64] is also a tree like structure that represents all the combinations of individual component states that cause the system to occupy a specific state. The main difference is that in MSFT, each system component can have many different states and not only two states as in binary-state system. The root of a tree is the top event, say S, which means that the system is in state S. The event S is reduced to a combination of events that can cause the occurrence of that particular top event by means of logic gates (e.g. *and, or*).

In a MSFT, a boolean variable is used to represent each state of the component. The variables associated with the same component are no longer independent of each other because the component can only occupy exactly one state at any time.

- **Example – MSFT**

Consider system with two boards B1 and B2, each having a processor and a memory. The memories (M1 and M2) can be shared by both processors (P1 and P2). This example has been taken from [64]. We consider the whole board as a component which has four states:

- State 4: both P and M are functional
- State 3: M is functional, but P is down.
- State 2: P is functional, but M is down.
- State 1: both P and M are down.

System state is defined as:

- State 'UP': at least one processor and both of the memories are functional.

Figure 2.5 shows the MSFT for state 3 of the system. In the figure 2.5 $B_{i,j}$ represents the board B_i being in state j.

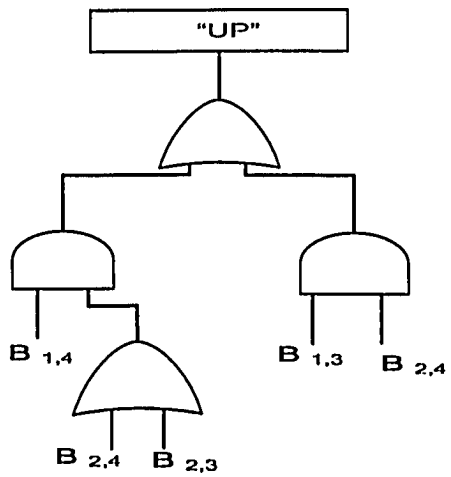

Figure 2.5 - A Multi State Fault Tree

MFST is used for dependability analysis part of the Rejuvenated-FTLQN model solution. Fault tree and MSFT can be solved using the SHARPE tool [47] [53].

2.6 Markov Chains

This section gives an overview of discrete and continuous time markov chains with the help of an example.

Stochastic Processes

A *Stochastic process* is a family of random variables $\{X\{\alpha\}, \alpha \in T\}$, where the parameter α takes values from the parameter set T. A stochastic process is a probabilistic model of a dynamic system where T is either a discrete or continuous representation of time. If T is discrete, then the events are observed at discrete points; if T is continuous, then events are observed continuosly over time. The random variables is a stochastic process take on values $X\{\alpha\} \in \Omega$, where Ω is called the *state space.*

If T is discrete set, the stochastic process is a discrete time stochastic process. If T is continuous set, the stochastic process is a continuous time stochastic process.

Markov Chains

A Markov process is a stochastic process which satisfies the markov (or memoryless) property: *the future of the process depends only on the current state of the process and the current time not on the history of the process.*

$$P(X(t) \geq x \mid X(t_i) = x_i, t \geq t_i \geq \ldots \geq t_n\} = P\{ X(t) \geq x \mid X(t_i) = x_i \} \qquad (1)$$

A *Markov chain* is a Markov process with a finite or countably infinite state space. At each observation, the process is seen to be in one of a countable number of states. These states are generally labeled as integers or vectors of integers [54].

A *Discrete Time Markov Chain (DTMC)* is a Markov process with finite or countably infinite number of states where the time parameter T is measure in countable units. It is denoted as $\{X_n , n \geq 0\}$. The Markov property can be stated as shown in equation *(2)*:

$$P\{X_{n+1} = j \mid X_n = i_n \ldots, X_o = i_o\} = P\{X_{n+1} = j \mid X_n = i_n\}; \qquad (2)$$

for $n \geq 0$ and $i_o, i_1 \ldots, i_n, j \in \Omega$

A *Continuous Time Markov Chain (CTMC)* is a Markov process with finite or countably infinite number of states where the time parameter T is continuous. It is denoted as $\{X(t) , t \geq 0\}$. The Markov property can be stated as shown in equation *(3)*:

$$P\{X(t+s) = j \mid X(s) = i, X(u) = K_u ; 0 \leq u < s\} = P\{X(t+s) = j \mid X(s) = i\} \qquad (3)$$

for $t, s \geq 0, i, j \in \Omega$, and for all $0 \leq u < s, k_u \in \Omega$.

Markov chains [54] have the notions of *state* of a system and *transitions* between states. The system is said to occupy a certain "state" whenever it satisfies the conditions defined for that state. The dynamical changes in the state of the system are referred to as "state transitions". In Markov model, it is assumed that the sojourn time (the amount of time spent in a state) is exponentially distributed. For dealing with non-exponentially distributed sojourn times, semi-Markov models were developed. The state space for the

Markov models grow much faster than the number of components being modelled, making it difficult to specify a model correctly.

- **Example-DTMC**

Suppose that if a particular machine remains broken for 4 days, the machine is replaced with a new machine. And for broken day 1, 2 and 3 the machine is repaired. "Broken" indicates the machine is in repair.

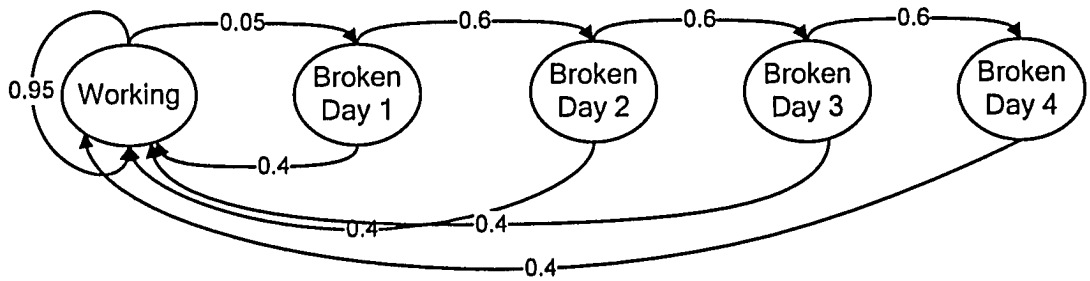

Figure 2.6 - Example-DTMC

- **Example-CTMC**

Consider a computer system that consists of two file servers and one processor. The system can function correctly as long as at least one of the file servers and the processor are operational. Both the file servers and processor can be repaired. The CTMC model showing the state transition is as shown in figure 2.7.

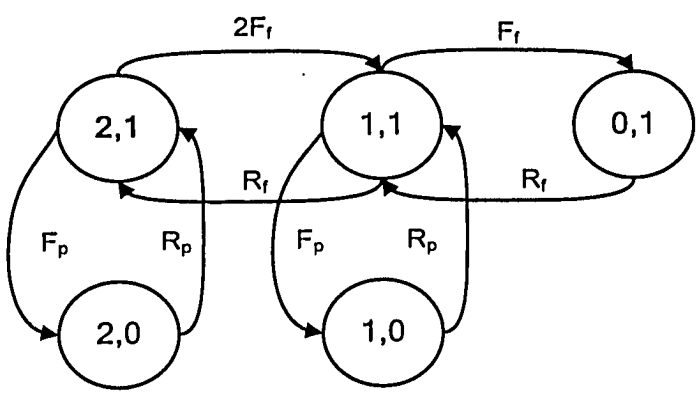

Figure 2.7 - Example-CTMC

- F_f - Failure rate of the file servers.
- F_p - Failure rate of the processors.
- R_f - Repair rate of the file servers.
- R_p - Repair rate of the processors.

A CTMC is used for modeling the effects of aging and rejuvenation in the Rejuvenated-FTLQN model solution.

2.7 Layered Distributed systems or Layered Architecture

The client-server computing paradigm is a system where processing of requests from users is distributed among several tasks. Tasks interact with one another using the remote procedure call (RPC) [60]. Applications make requests for services using what appear to be conventional procedure calls. However, rather than branching to another section of the same program, a message is sent to another task which may or may not reside on the same computer. When the remote task replies, the remote procedure call returns.

Distributed software systems are usually structured in layers, with some kind of operational control or user interface tasks as the topmost layer, making requests to various layers of servers. Layered modeling describes a system by the sets of resources that are used by its operations. Every operation requires one or more resources, and the model defines a resource context and an architecture context for each operation. The architecture context is a software object to execute the operation, and the resource context is a set of software and hardware entities required by the operation. Every resource includes an aspect of an authority to proceed and use it, which is controlled by a discipline and a queue (which may be explicit or implicit). In layered modeling the resources are ordered into layers (typically with user processes near the top and hardware at the bottom) to provide a structured order of requesting them. With layering a graph of all possible sequences of requests is acyclic, and deadlock among requests is impossible. Layering provides an order; requests may jump over layers.

2.8 Queueing Network and Layered Queueing Network (LQN) Model

This section gives an overview of Queueing Network and its extension Layered Queueing Network (LQN) model. Consider a *service center* and a *population* of *customers*, which at some times enter the service center in order to obtain service. It is often the case that the service center can only serve a limited number of customers. If a new customer arrives and the service is exhausted, he enters a *waiting line* and waits until the service facility becomes available. So we can identify three main elements of a service center:

1. Population of customers,
2. Service facility and
3. Waiting line.

As a simple example of a service center consider an airline counter where passengers are expected to check in, before they can enter the plane. The check-in is usually done by a single employee, however, there are often multiple passengers. A newly arriving and friendly passenger proceeds directly to the end of the queue, if the service facility (the employee) is busy. This corresponds to a FIFO service (first in, first out).

Queueing network modelling is a particular approach to computer system modelling in which the computer system is represented as a network of queues which is evaluated analytically. A network of queues is a collection of service centers, which represent system resources and customers, which represent users or transactions [35]. Thus queueing network is a pure performance model.

LQN (pure performance model) is an extension to the widely used queueing network model. LQN was first independently developed under the name of Stochastic Rendezvous Networks (SRN) in [22] [45] [60]. The most important difference between LQN and traditional queueing networks is the fact that a server serving a client request can become a client of another server, thus modelling nested services and synchronous calls. This way, a concept of layering is introduced.

The Layered Queueing Network is a model of network of tasks running on processors and communicating via a send-reply-receive pattern, in which a sender of a message waits for a reply pattern called as Rendezvous, a RPC, or synchronous messaging. The task may also send messages without reply, known as asynchronous messaging. Calls can target servers in the same layer as the client or can skip several layers [21].

LQN Building Blocks

This section describes the main components of the LQN Model with the help of an example. The definitions in this section are taken from [21].

- **Task**

A task is an entity that models a software process. A task can be either:

1. Client task
2. Server task

A client task sends requests to other tasks. A server task performs work on behalf of the request from its clients. A server itself may also be a client to its lower level servers by making requests to those as part of fulfilling their own work to the higher level client. Each task may have different classes of workloads on the processor by representing it with several entities. Each entry provides a different service pattern and a different workload. However, all entries of one task share a common task queue. The task queue scheduling disciplines supported by LQN that controls the order in which requests are processed E.g. First In First Out (FIFO).

A server may be a single server, a multiserver or an infinite server. A single server is modeled as a single task, which handles only one request at a time. Concurrency in LQN is modeled by multi-servers and replicated servers. A multiserver contains a multitude of copies of task, yet all copies share one common queue for incoming requests. A replicated server, however, is similar to the multiserver, except that each copy task has its own request queue. An infinite server is modeled as an infinite number of processors that

can handle an infinite number of requests. For example, network delays are often modeled as infinite servers.

- **Entry**

An LQN server may offer more than one service, each one with it own service time and visit ratio to other servers. Each service is modeled as an entry of the task. It is assumed that all the requests for all entries of a task are queued in a common task queue.

The execution of a server entry following the reception of a message by an entry maybe broken into two parts; the first part named first phase ends when the reply is sent back and the second part is the subsequent phases after the reply.

- **Types of Request**

1. **Synchronous Request-** A Synchronous Request blocks the client until server sends back the reply.
2. **Asynchronous Request-** A client continues its work in parallel with the server.
3. **Forwarding Request-** A Forwarding Request is similar to synchronous request from the client's point of view. The difference is that more then one server is involved. The first server forwards the request to the next server, and it is free to do other work, after the second server finishes the request, the second server sends back the reply to the original client. The original client is blocked until it receives the reply.

In the LQN model shown in figure 2.8, we have "console task" (reference task), "application task", "generate task" and "database task".

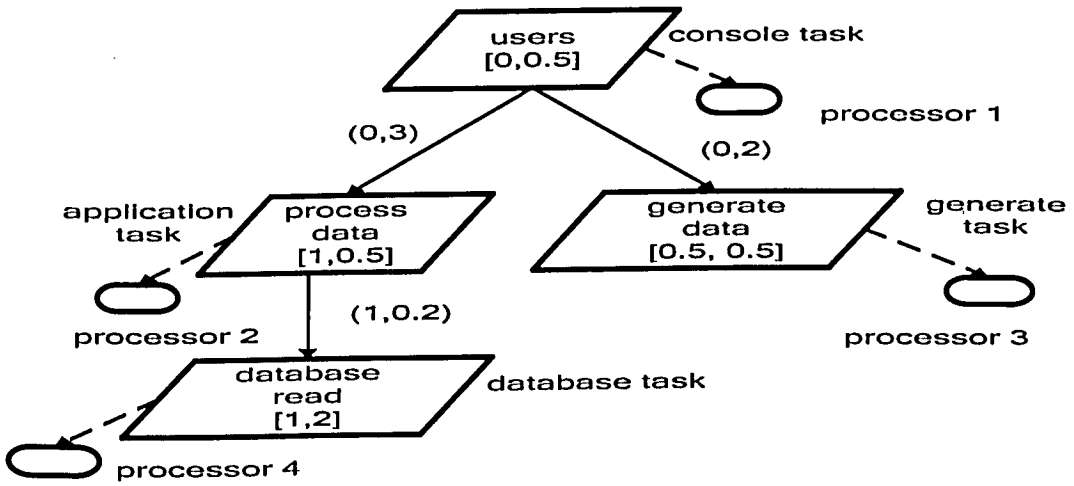

Figure 2.8 - An LQN Model

Tasks can provide more than one type of service (e.g. a database can provide support for both searching and updating). This is represented by dividing the task-parallelogram into several smaller parallelograms (called *entries*), each representing one type of service. Request-arcs are then drawn to the sub-parallelograms. Every task has at least one entry. Thus each kind of service offered by an LQN task is modeled as a so-called *entry*. In figure 2.8, "users", "process data", "generate data" and "database read" are entries. An entry may be decomposed in two or more sequential phases of service. Phase 1 is the portion of service during which the client is blocked, waiting for a reply from the server. At the end of phase 1, the server will reply to the client, which will unblock and continue its execution. The remaining phases, if any, will be executed in parallel with the client. Every phase has its own execution time and demands for other entries. The values specified in the square brackets indicate mean execution demand for one invocation of entry. The values specified near the arrows indicate the average number of calls to other entry, for one invocation of the entry. In figure 2.8, for e.g. [1, 0.5] indicates execution demand of entry "process data" and (0, 3) indicates the average number of calls made by entry "users" to "process data". Although not explicitly illustrated in the LQN notation, every server, be it software or hardware, has an implicit message queue where incoming requests are waiting for their turn to be served. Servers with more than one entry have a single input queue, where requests for different entries wait together.

2.9 Fault Tolerant Layered Queueing Network (FTLQN) Model

This section describes the FTLQN model, which is an extension of the LQN model. An LQN model is a pure performance model, and FTLQN adds dependability-related parameters to it. An FTLQN [9] model describes distributed systems which employ redundant servers which may be primary-backup or load-balanced. It modifies the LQN model to express the strategy to be used in case of failure and to generate the different configurations in which the system may be fully or partly operational. The layered structure of an FTLQN model describes the dependencies that determine the service failures, based on service dependencies.

In an FTLQN model, a service request arc may be replaced by a set of alternative arcs with an order of preference. In Figure 2.9, we have database task-A as well as database task-B. So if database task-A (having priority 1 indicated by #1) fails then application task will use database task–B (having priority 2 indicated by #2). A set of alternative arcs are shown attached to *solid black rectangle.*

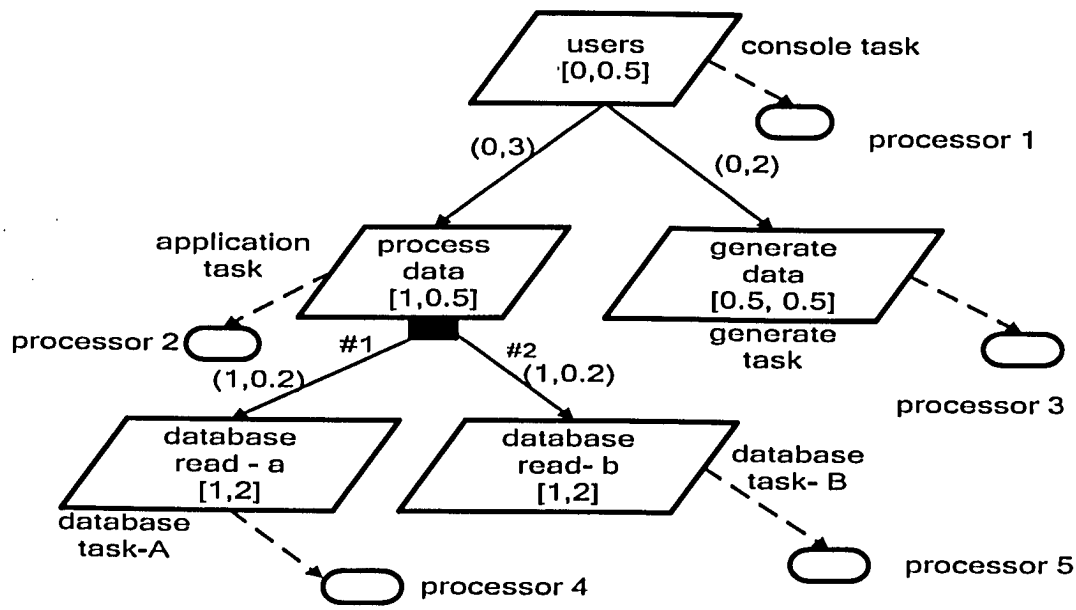

Figure 2.9 - An FTLQN Model

The FTLQN model has two advantages [10]:

- It closely resembles the software architecture (making it easy to build).
- It contains the service dependencies which are required in one form or another to analyze the failures.

Thus FTLQN model extends the LQN performance model by adding dependability-related parameters to it.

2.10 Different Modeling Approaches for Software Rejuvenation

This section gives an overview of two different modeling approaches for software rejuvenation: (1) Analytical modeling and (2) Measurement based approach.

Analytical modeling

Figure 2.10 shows the basic software rejuvenation model proposed by Huang et al. [31]. The software system is initially in a "robust" working state, 1. As time progresses, it eventually transits to a "failure-probable" state 2. The system is still operational in this state but can fail (move to state 0) with a non-zero probability. The system can be repaired and brought back to the initial state 1. The software system is also rejuvenated at regular intervals from the failure probable state 2 and brought back to the robust state 1.

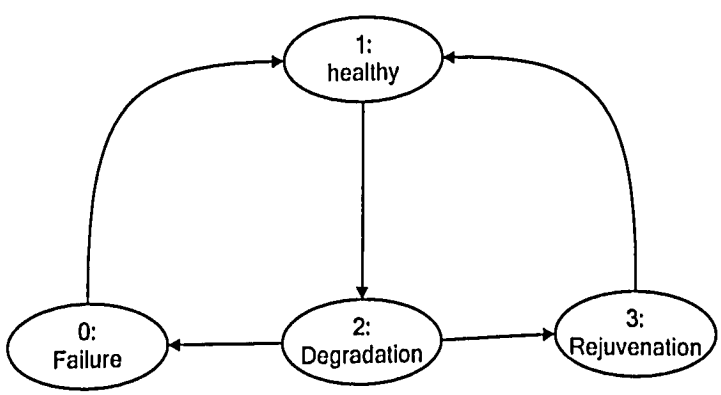

Figure 2.10 - State Transition Diagram

Huang et al. [31] assumed that the stochastic behavior of the system can be described by a simple Continuos-Time Markov Chain (CTMC). The random time interval when the highly robust state changes to the failure probable state is exponentially distributed. Just after the state becomes the failure probable state, a system failure may occur with a positive probability. If the system failure occurs before triggering software rejuvenation, then the repair is started immediately at that time and is completed after the random time elapses. Otherwise, the software rejuvenation is started. Note that the software rejuvenation cycle is measured from the time instant just after the system enters state 1.

Dohi et al. [19] developed semi-Markov models with the periodic rejuvenation and general transition distribution functions. The underlying stochastic process is a semi-Markov process with four regeneration states. If the sojourn times in all states are exponentially distributed, this model is the CTMC in Huang et al. Thus the model has similar but somewhat generalized mathematical structure to that compared to Huang et al.

Garg et al. [24] developed a Stochastic Petri Net (SPN) model where rejuvenation is performed at deterministic intervals. Because of the deterministic rejuvenation intervals the theory of CTMC cannot be applied. The software starts up in a "robust" state in which the probability of failure is zero. As it is used, it ages with time and if no rejuvenation is done eventually transits to another state. In this state, it provides normal service but can fail (crash) with a non-zero probability. Once it crashes, it takes a random amount of time to bring it up again to the clean state and restart it. Rejuvenation is performed at a fixed interval from the start (or restart) of the software in the robust state. At the time of rejuvenation, if the software has not already crashed, it is either in the clean or the failure probable state. It is then stopped, cleaned and restarted; all of which takes a random amount of time. Further it was assumed that the time for which software remains clean and the time to fail from the failure probable state are both exponentially distributed. Thus the time to failure for the software starting in the robust state has a hypo-exponential distribution. The times to restart both from rejuvenation and crash failures are both exponentially distributed. The rejuvenation interval, however, is deterministic.

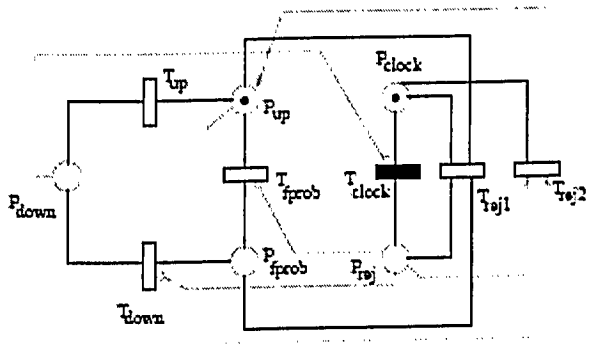

Figure 2.11 – Petri Net Model for Software Rejuvenation

Figure 2.11 shows the Petri net model for software rejuvenation [24]. The circles represent, places with dots inside representing the tokens held inside that place. Unshaded rectangles represent transitions with exponentially distributed firing time while the shaded rectangle represents a transition with a constant firing time. The robust state is modeled by the place P_{up}. Transition T_{fprob} models the aging of the software. When this transition fires, i.e., a token reaches place P_{fprob}) the software enters the failure probable state. The transition T_{down} models crash failure of the software. During the software restart (while the transition T_{up} is enabled), every other activity is suspended; the inhibitor arc from place P_{down} to transition T_{clock} is used to model this fact. The transition T_{clock} models the rejuvenation period. It is enabled with T_{fprob} and fires when the clock expires if T_{fprob} has not fired by that time. Once it fires, a token moves in place P_{rej} and the activity related with software rejuvenation (transition T_{rej}) starts. During the rejuvenation phase, every other activity in the system is suspended. This is modeled by inhibitor arcs from place P_{rej} to transitions T_{fprob} and T_{down}. Upon rejuvenation, the Petri net has to be reinitialized into a condition with one token in place P_{up} and one in place P_{clock}, and all the other places empty.

Wang et al [59] considered different rejuvenation policies for clustered system with n identical nodes and used Petri Nets for modeling purpose. Three rejuvenation policies evaluated were: (1) *Standard rejuvenation*- the rejuvenation is triggered after T time units have passed since the last rejuvenation epoch or the recovery from system failure. (2)

Delayed Rejuvenation- In off peak period, the rejuvenation policy is the same as standard rejuvenation policy. In peak period, all nodes are just scheduled for rejuvenation if time T has passed since last rejuvenation. Nodes scheduled for rejuvenation still operate as usual until rejuvenation starts immediately when next off peak period starts. (3) *Mixed Rejuvenation*- This policy combines the standard rejuvenation and the delayed rejuvenation policies. In off peak period, the rejuvenation policy is the same as standard rejuvenation policy. In peak period, the rejuvenation will be done if scheduled early in the peak period, while it will be delayed if scheduled late in the peak period. According to [57], the delayed rejuvenation is better than the standard rejuvenation with respect to the system throughput. For longer rejuvenation-triggering intervals, the standard rejuvenation yields a better result than delayed rejuvenation, while for shorter rejuvenation-triggering intervals the delayed rejuvenation policy outperforms standard rejuvenation policy. While mixed rejuvenation policy provides better results compared to both standard and delayed rejuvenation policies.

Measurement based approach

While the analytical model is based on the assumption that the rate of software aging is known, in the measurement based approach, the basic idea is to monitor and collect data on the attributes responsible for determining the health of the executing software. The data is then analyzed to obtain predictions about possible impending failures due to resource exhaustion.

The basic idea is to periodically monitor and collect data on the attributes responsible for determining the health of the executing software. Garg et al. [25] propose a methodology for detection and estimation of aging in the UNIX operating system. An SNMP-based distributed resource monitoring tool was used to collect operating system resource usage and system activity data from nine heterogeneous UNIX workstations connected by an Ethernet LAN. A central monitoring station runs the manager program which sends *get* requests periodically to each of the agent programs running on the monitored workstations.

2.11 Performability: Measures and Models

In the past, most modelling work kept performance and dependability separate. Initially, the dependability of the system might have been satisfied, then the performance optimized. This led to systems having good performance when the system was fully functional but a drastic decline in performance when, inevitably, failure occurred. Basically, the system was either 'on' and running perfectly or 'off' when it crashed. Improvements on this led to the design of degradable systems. Because degradable systems are designed to continue their operation even in the presence of component failures (albeit at a reduced performance level), their performance can not be accurately evaluated without taking into account the impact of the structural changes (malfunctions & repairs) [38].

Analysis of the systems from a pure performance viewpoint tended to be optimistic since it ignores the failure-repair behaviour of the systems. At the other extreme, pure dependability analysis tended to be conservative, since performance considerations were not taken into account. Thus, it was essential that methods for the combined evaluation of performance and dependability be developed: Performability analysis.

2.11.1 Performability Measures

The goal of evaluating the performability of a system is to capture the overall quality of its responsiveness which is sometimes degraded by failures. Performability was first introduced by [38] as a measure that quantifies a system's ability to perform in the presence of faults. Such ability has been formally defined as the probability that the system will perform above a given accomplishment level, given that the system has been operational for a period of time t. This section defines various performability measures of interest as described in [30] [47].

(1) ***Steady-state performability or the expected steady-state reward rate***:

$$P_{steady} = \sum_{i \in \chi} P_i \, R(i) \qquad (4)$$

Steady state performability is defined by equation *(4)*, where:

- P_{steady} indicates the steady state performability or expected steady-state reward rate.
- χ denotes the set of all the possible configurations in which the system can operate.
- P_i stands for the steady state probability of states residing in χ.
- R(i) stands for the reward rate associated to the states in χ .This reward rate quantifies the ability of the system to perform in the corresponding configuration.

We can then partition the set of states χ in a set χ_u of "up" states, and a set χ_d of "down"states, i.e. $\chi = \chi_u \cup \chi_d$. with $\chi_u = \{ i \in \chi \mid R(i) > 0 \}$ and $\chi_d = \{ i \in \chi \mid R(i) = 0 \}$. Consequently we then have,

$$P_{steady} = \sum_{i \in \chi} P_i R(i) = \sum_{i \in \chi_u} P_i R(i) \qquad (5)$$

In a particular case, when all the reward rates R(i) = 1, whenever $\chi \in \chi_u$ and R(i) = 0 elsewhere, then the above equation (5) gives the steady-state dependability D as shown in equation (6):

$$D = \sum_{i \in \chi} P_i \qquad (6)$$

(2) ***Point Performability or the expected instantaneous reward rate is given by***:

$$P_{instant}(t) = \sum_{i \in \chi} P_i(t) R(i) \qquad (7)$$

Point performability is defined by equation *(7)*, whenever $R(i) = 1$ for all the system operational states and zero otherwise, the above equation gives the system instantaneous dependability.

(3) ***Cumulative Performability***:

The cumulative performability or the accumulated reward in [0, t) which represents the

amount of work accomplished during a generic time interval [0, t) is given by equation *(8)*:

$$Y(t) = \int_0^t Z(t)\, dt \qquad (8)$$

Where $Z(t)$ denotes the system reward rate at time t i.e. the reward value of the state of the system at time t.

(4) ***Performability Distribution***:

The performability distribution or the distribution of the accumulated reward (denoted by $F(t, y)$) is given by equation *(9)*:

$$F(t, y) = Prob\{\ Y(t) \leq y\ \} \qquad (9)$$

(5) ***Mean accumulated reward***:

The mean accumulated reward in [0, t) can be calculated using equation *(10)*:

$$E[Y(t)] = E[\int_0^t Z(\tau) d\tau] = \int_0^t E[Z(\tau)]\, d\tau \qquad (10)$$

The fraction $E[Y(t)]/t$ is known as the interval availability when a reward rate 1 is assigned to the system operational states and zero to non-operational states.

2.11.2 Performability Models

The common approach for the development of performability models of fault-tolerant computer systems has been the use of *stochastic reward models [30]*. In a stochastic reward model there is a *stochastic model* (also known as the structure state model) which describes the configurations of the system (i.e. structural variations of the system arising due to the failure and repair of its components) and a *reward rate* (measure of performance) which is associated with each of the states of the stochastic model. The stochastic model is thus the higher level dependability model representing the failure/repair processes of the system components. The performability measure is obtained by combining the reward rates associated with the states of the stochastic model with the state probabilities.

(1) Markov Reward Models (MRM)

In a Markov Reward Model, the underlying structure state model is a Markov model (a CTMC). In the Markov model, the sojourn time (the amount of time in a state) is assumed to be exponentially distributed. A reward rate (measure of performance) is associated with each states of the Markov model and the desired performability measures can be obtained by combining the reward rate associated with each state of the Markov model with the state probabilities.

The first issue related to the Markov Reward Model is concerned with the description of the dependability aspects of a system, i.e. the translation of the system into its corresponding Markov model. The Markov model has some associated difficulties [47]. The state space can grow much faster than the number of components in the system being modelled, making it difficult to specify a model correctly. The next issue related to Markov Reward Model is concerned with the derivation of the reward rates. Such derivation usually requires a performance evaluation in each state of the Markov model. The performance evaluation in each state of the Markov model may be measurement based or model based.

(2) Dynamic Queueing Network

A version of the Markov Reward Modelling approach known as Dynamic Queueing Network concept has been developed [30] in which a parameterized queueing network is used to model the performance aspect of the system and a generalized stochastic Petri net is used to describe the dependability aspect of the system. This technique comprises of a function that maps the possible markings of the generalized stochastic Petri net, denoting the system configurations, to the queueing network parameters. Then, a reward rate is associated with every possible generalized stochastic Petri net markings by solving the corresponding queueing network models. Finally, the actual calculations for performability are done by translating the dynamic queueing network model to a Markov Reward model. The performability modeling approach taken in this thesis is similar to the dynamic queueing network concept.

(3) Non-Markov Reward Models

In a Markov Reward Model, the amount of time in a state is assumed to be exponentially distributed. There are a great number of real situations in which the failure processes are non-Poisson and the failure times of the components are deterministic or generally distributed. For dealing with non-exponentially distributed failure times, the semi-Markov Reward Model has been developed. In a semi-Markov Reward Model, the underlying structure state model is a Semi-Markov process (SMP) in which the rate of transition from state i to j may depend on how long the chain has been in state i, but it still does not depend on anything that happened before the chain reached state i. In SMP, the amount of time in a state is allowed to be any distribution function [30].

Chapter 3

Rejuvenated-FTLQN Model and its Solution

This chapter describes the "Rejuvenated-FTLQN" model and its solution. The main purpose is to evaluate the performability of fault tolerant layered distributed systems under the effects of software aging and rejuvenation. The goal of evaluating the performability of a system is to capture the overall quality of its responsiveness which is sometimes degraded by failures [38]. In this thesis, the service degradation caused by software aging is also considered. A model called "**Rejuvenated-FTLQN**" is proposed for this purpose.

In section 3.1 of this chapter, the Rejuvenated-FTLQN model is described, in section 3.2 the Rejuvenated-FTLQN model solution is introduced and in section 3.3, each and every step of the model solution is explained with the help of an example.

3.1 Rejuvenated-FTLQN Model

An FTLQN model [9] describes distributed systems which employ alternative servers and routing of requests to mask failures. FTLQN model modifies the LQN model to express the strategy to be used in case of failure. Rejuvenated-FTLQN model adds software aging and rejuvenation related information to FTLQN model. This section gives an overview of the Rejuvenated-FTLQN model and the next section describes its solution.

3.1.1 System-level Model

Figure 3.1 shows an example of FTLQN model of the system. The alternative targeting of the request is indicated by the labels "#n" on arcs showing the priority of the targets.

#1 indicates the highest-priority available server. A task is an entity that models a software process. T1, T2, T3, T4 and T5 indicates software tasks (processes). A server task performs work on behalf of the request from its clients. A server itself may also be a client to its lower level servers. A task may offer different services. Each service is modeled as an entry of the task. In figure 3.1, e1, e2, e3, e4 and e5 indicates entries.

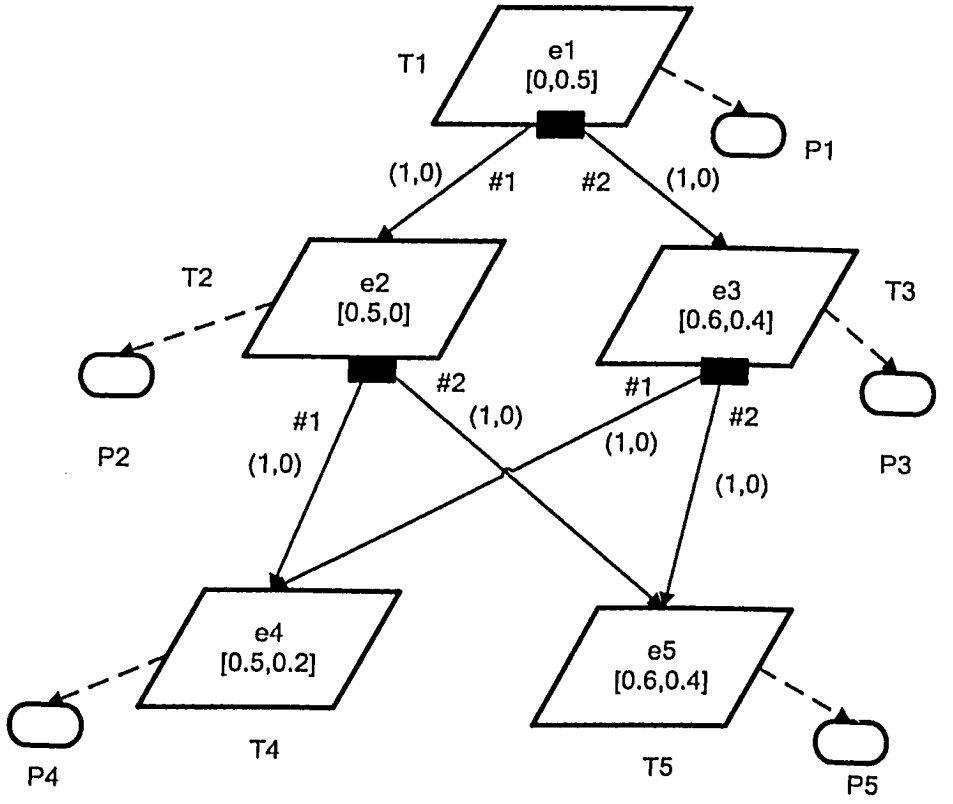

Figure 3.1 – Example- FTLQN Model

The performance parameters provided in this model are:

- The mean CPU demand per invocation foe each entry (E.g. [0.5, 0] inside the parallelogram).
- The mean number of calls from an entry to other entries (E.g. (1,0) near the arrow)

3.1.2 Model for Individual Components

In this section, the model considered for taking into account the effects of aging and rejuvenation on the software tasks is described. Huang, et al. [31] proposed an analytical model for modeling aging and rejuvenation, where the degradation is described by a two step process. From the clean state, the software system makes a transition into a degraded state from which two actions are possible: rejuvenation with return to the clean state or transition to the complete failure state. They model the four-state process as a Continuous-Time Markov chain (CTMC). Each of the four states is described as follows:

- **State W**: Working or highly robust state (normal operation)
- **State FP**: Failure Probable state (due to aging).
- **State BR**: Being Rejuvenated state (undergoing rejuvenation)
- **State F**: Failed State.

Figure 3.2 shows the state transitions for the software tasks without rejuvenation. An application stays "healthy" for a while before it reaches a state where failure is probable; it often takes a while for a program to reach its boundary conditions or leak out some of its resources [31]. Thus a failure is two step behavior as shown in figure 3.2, where a process goes from state 'W' (highly robust state) to state 'FP' (failure probable state) with rate R_2 (the time interval is called the *base longevity interval*) and from there it can make a transition to state 'F' (failed state).

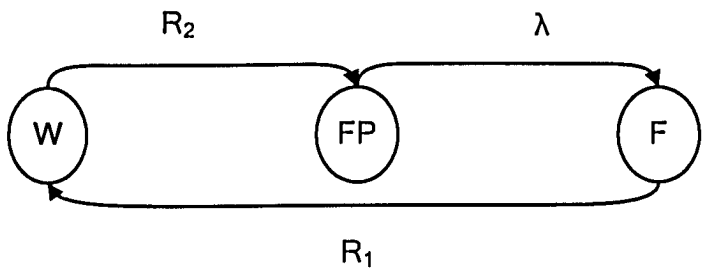

Figure 3.2 – CTMC Model for Software Task– Without Rejuvenation

Now consider the CTMC for the software tasks with rejuvenation implemented as shown in figure 3.3. In this case we have an additional state (State 'BR') which indicates that the task is undergoing rejuvenation. Hence a process can make a transition from State 'FP' (Failure Probable state) to State 'BR' (Being Rejuvenated) state with rate R_4. After rejuvenation is performed the process goes back to State 'W' (highly robust state) with rate R_3.

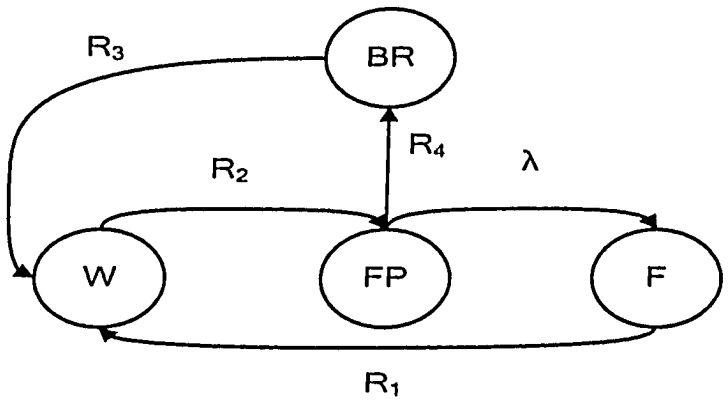

Figure 3.3 - CTMC Model for Software Task – With Rejuvenation

The CTMCs described above is solved to get the steady state probabilities of the software tasks being in each of the four states. Figure 3.4 shows the CTMC model for the processor. Aging and rejuvenation are not considered for processors.

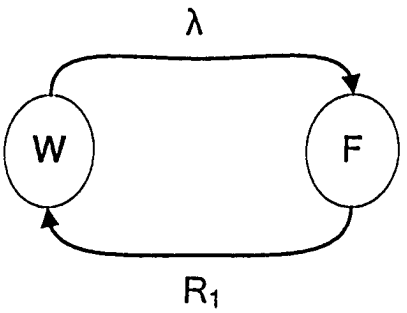

Figure 3.4 - CTMC Model for Processor

3.1.3 Modelling Fault Propagation

For modeling fault propagation, the FTLQN model is converted into the fault propagation AND-OR graph [10]. AND-OR graphs are used for problem reduction by decomposing the main problem into the set of sub problems. AND-OR graphs can have *and* nodes whose successors must *all* be achieved, and *or* nodes where *one* of the successors must be achieved (i.e., they are alternatives).

The main goals of the fault propagation AND-OR graph are:

- To model the service dependencies in the FTLQN model and
- To generate all the operational configurations (different configurations of the system in which the system is operational). For this purpose the graph is traversed in Breath First Search (BFS) fashion.

The fault dependencies in an FTLQN model can be represented by a prioritized, labeled directed AND-OR graph termed as a *fault-propagation AND-OR graph*. Because of the layered structure of an FTLQN model, the failure of a service provided by an entry depends on the failure of the services provided by its processor, its task and by other lower-level entries in the model. Thus the entry *e* of task *t* is said to be *operational* if:

1. The task *t* is working. Task can be any of the two states: State 'W' (highly robust state) or State 'FP' (failure probable due to aging).
2. The processor allocated to task *t* is working.
3. All the services that the entry *e* uses are working. A service used by an entry is said to be working if any of the alternative entries providing that service is working.

The entry *e* is *unoperational* otherwise. The dependencies on task *t* and its processor are represented separately because a failure of task *t* may be independent of the failure of its processor.

Graph contains five types of nodes, namely, *entry node, service node, processor node, task node* and *state classification node* which are defined as follows:

(i) An *entry node* is an AND node that describes an entry of the FTLQN model. Figure 3.5 shows the graph for the FTLQN model shown in the figure 3.5. In Figure 3.5- e1, e2, e3, e4 and e5 are all entry nodes.

(ii) A *service node* is an OR node that describes the preference order of the alternative targets for a requested service. All of its successors represent the alternative targets while its parent represents the entry requesting the service. This node corresponds to the solid black rectangle in the FTLQN model and is represented as a shaded node in graph. In figure 3.5- Serv1, Serv2 and Serv3 are service nodes.

(iii) A *processor node* is a leaf node that contains the information about a processor. In figure 3.5 P1, P2, P3, P4 and P5 are processor nodes.

(iv) A *task node* contains the information about a software task of the FTLQN model. Graph also has one special start node r which is an OR node representing the overall state of the system. Its successors represent the reference entries of the FTLQN model, a reference entry being an entry of a reference task. In figure 3.5- T1, T2, T3, T4 and T5 denotes task nodes.

(v) A *state classification node* contains the information about the different states that the software task can possess when it is operational. For example it may be in State 'W' (highly robust state) or State 'FP' (failure probable state- due to aging). In figure 3.5- 'W' and 'FP' indicates the two operational states of the software tasks.

In the FTLQN performability algorithm as described in [9], the fault propagation And-Or graph representation of FTLQN model consists of all the above mentioned nodes except the state classification node. In Rejuvenated-FTLQN model, the software tasks can be in 'W' or 'FP' state, a state classification node is added to make this distinction.

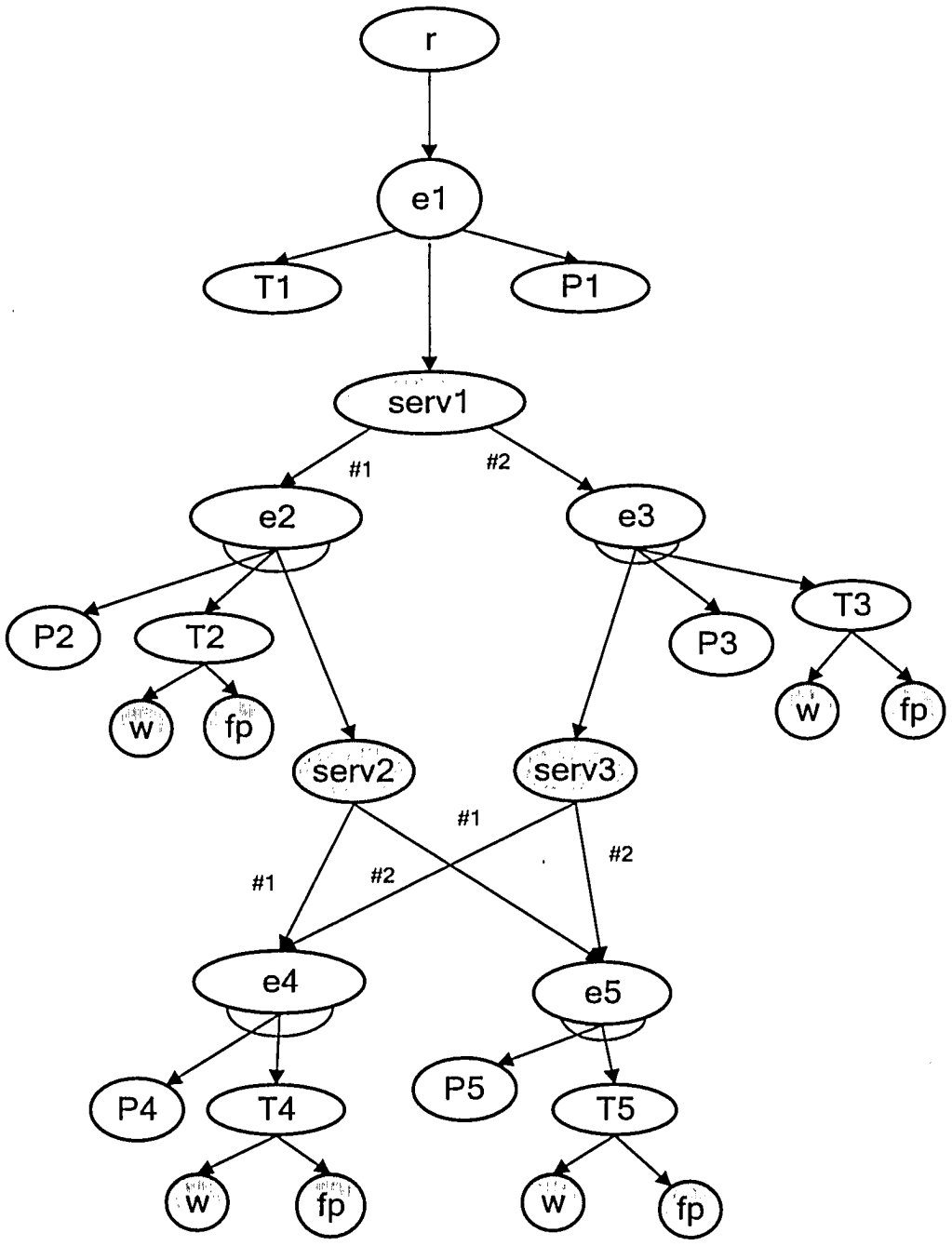

Figure 3.5– Fault Propagation And-Or Graph for FTLQN Model

3.1.4 Operational Configurations

Structural variations of the system arising due to the failure of its components and software aging describe the configurations of the system. The layered structure of an FTLQN model describes the dependencies that determine the service failures, based on

service dependencies. The special property of layered systems is that a failure of a task or processor in one layer can cause many tasks that depend on its services (at any layer in the system) to fail, unless there is a redundant server. This property gives rise to the different configurations of the system in which the system is operational. All this set of configurations is defined as *operational configurations.*

For every software task to be operational it can be in State 'W' (highly robust state) or in State 'FP' (failure probable state due to software aging). This behavior gives rise to even more configurations. We are interested in capturing all the different combinations of the system components (tasks) states that lead to the whole system being operational.

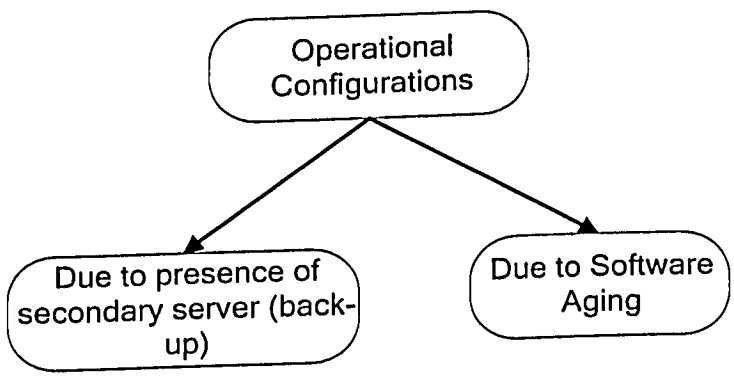

Figure 3.6– Different Operational Configurations

3.1.5 Modelling Performance Degradation due to Software Aging

When the task is in 'W' state the mean CPU demands for its entries will be lower compared to when it is in 'FP' state. Aging of the task is modeled by making the mean CPU demands or mean execution demands of its entries an increasing function of its state. When the task is in "FP" state its performance is degraded and the mean CPU demands for all of its entries will be increased as per the rate at which the service degradation occurs. E.g. Performance of primary database server is degraded by 40% due to software aging, so from the end-user perspective it will take more time to process the request (compared to when it was in highly robust state) resulting in increased response

time. To model this behaviour we are increasing the execution demands (mean CPU demands) for the corresponding entries in the failure probable state. We can do this because the service rate reaches and settles to a low unacceptable value as a result of software aging [44]. Different LQN models that correspond to different operational configurations will differ in the parameters as per the state the software tasks resides in ('W' or 'FP').

3.1.6 Measure of interest

The goal is to evaluate the steady state performability of the system which is defined as:

$$P_{steady} = \sum_{i \in \chi} P_i R(i)$$

- χ denotes the set of all the possible configurations in which the system can operate
- P_i stands for the steady state probability of states residing in χ
- R (i) stands for the reward rate associated to the states in χ
- P_{steady} denotes the steady state performability

The reward rate quantifies the ability of the system to perform in the corresponding configuration. Thus P_{steady} represents the value of the reward rate offered by the system averaged over all the possible values it can provide according to the states it is in.

3.2 Rejuvenated-FTLQN Model Solution

To evaluate the steady state performability of the fault–tolerant layered distributed systems under the effects of software aging and rejuvenation following steps can be performed:

Step 1: Generate a FTLQN model from the system description. An FTLQN model describes distributed systems which employ redundant servers which may be used as a primary-backup. It modifies the LQN model to express the strategy to be used in case of failure.

Step 2: Add a Continuous Time Markov Chain (CTMC) for every software task (process) to the FTLQN model to take into account the effects of software aging and rejuvenation. CTMC denotes the *state* of a software task and *transitions* between states.

Step 3: Translate the FTLQN model into the Fault Propagation AND-OR graph. The Fault Propagation AND-OR graph provides a convenient means to model the dependencies of the service failures in a layered system. The Fault Propagation AND-OR graph also takes under consideration the multiple states of the software tasks resulting due to the performance degradation caused by software aging.

Step 4: Determine the different operational configurations of the system. Presence of secondary server and software aging gives rise to the different configurations of the system in which the system is operational.

Step 5: Evaluate the reward rate for each operational configuration using Layered Queuing Network (LQN) model for every configuration. Reward rate is the performance measure that will be assigned to every operational configuration to quantify how well the system performs in that particular operational configuration.

Step 6: Compute the probabilities of system being in each of the operational configuration. Multi-State Fault Tree's (MSFTs) is used to compute the probabilities. The input to the MSFT is obtained by solving the CTMCs for the software tasks as well as failure probabilities for processors.

Step 7: Combine the probabilities and the rewards for every operational configuration to determine the steady state performability.

All the above mentioned steps are automated in the software tool called **Rejuvenated-FTLQNS** (Rejuvenated - Fault Tolerant Layered Queuing Network Solver). Detailed explanation of the tool is given in chapter 5 of this thesis.

3.3 Example- Rejuvenated-FTLQN Model Solution

This section explains in detail all the steps mentioned above with the help of a simple example.

Step 1: Model Description

Consider an example of customer-information retrieval system, as described below:

1. User enters their credentials into the login screen.
2. User is authenticated and redirected to the search window.
3. User enters the search information (Name, ID etc) in the search window and clicks "Search".
4. Required record is fetched from the database and displayed to the user.

The FTQLN model for above mentioned system is as shown in figure 3.7, secondary database-2 task will be used by the application task whenever primary database-1 task is not available. Users task is the reference task. Interface task consist of one entry named

"Authenticate" having mean execution demand of [0.2, 0.1] and its processor named "Interface_P". Similarly for Application task, Database-1 and Database-2 task, we have entries "search", "db_read1" and "db_read2" respectively. The mean execution demand for every entry is as shown in figure 3.7. The processors corresponding to every task is also shown in the figure 3.7. The value (1, 0) near the arrow indicates the average number of calls made by one entry to another.

This example will be used to explain all the further steps of the model solution.

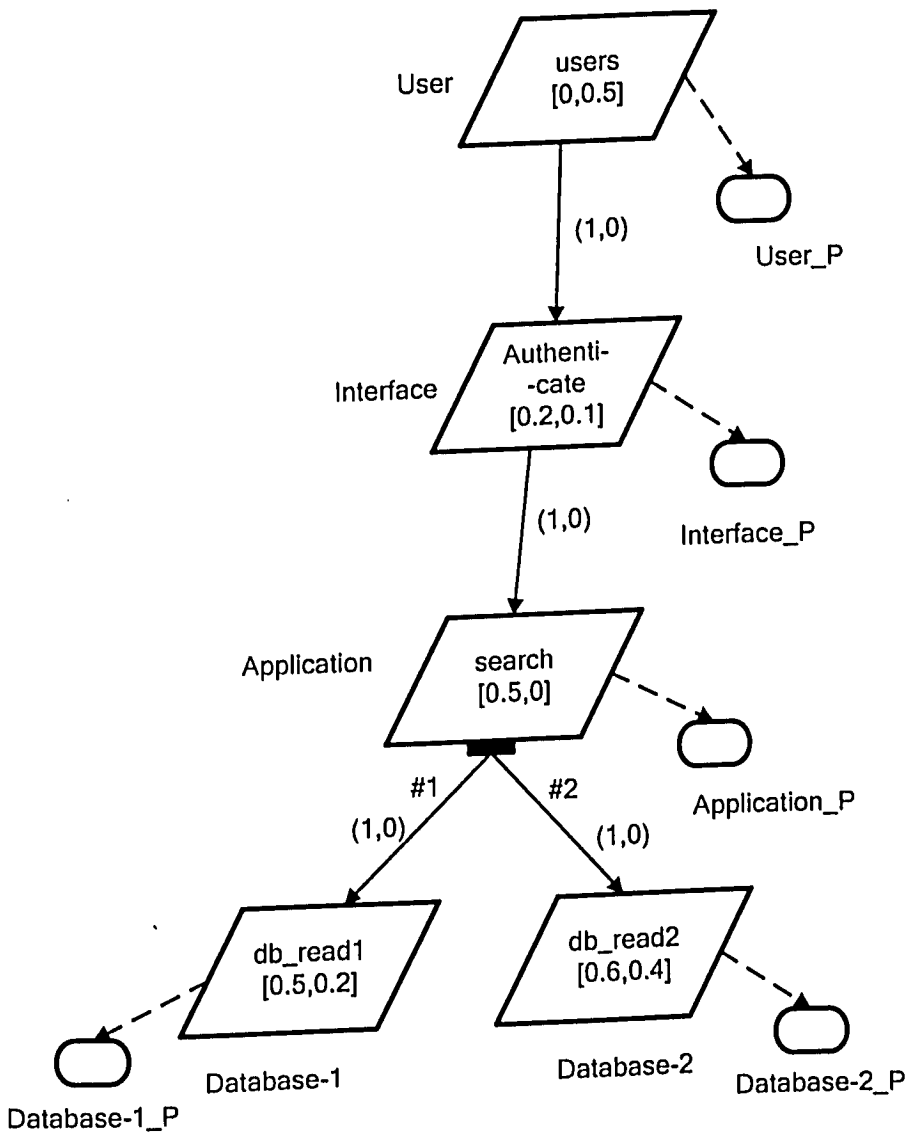

Figure 3.7 - FTLQN Model

Step 2: Modeling Aging and Rejuvenation for Software Tasks

In figure 3.7, Users task is a reference task (do not receive any request) and represent users (or load generators) of the system. So we don't consider aging and rejuvenation for that task. But for all the other tasks involved in customer information retrieval scenario the model parameters (assumed) are as follows:

Model Parameters

Interface task:

- It goes from initial robust state ('W') to failure probable state ('FP') in 5 days; $R_2 = 1/ (5*24)$.
- The mean time between two consecutive failures is 3 months; $1 = 1/ (3*30*24)$.
- To recover from an unexpected failure it takes 1 hour; $R_1 = 1$.
- Rejuvenation is performed every 2 weeks; $R_4 = 1/ (9*24)$. Note that R_4 denotes the rate of rejuvenation after the application goes into the failure probable state. Therefore we have $R_4=1/ \{(14 -5)*24\}$.
- Time to perform rejuvenation is 10 minutes. $R_3=6$.

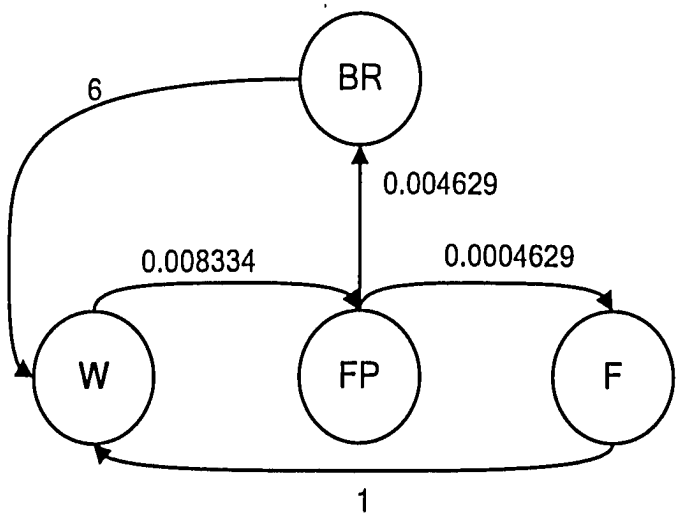

Figure 3.8 - CTMC –Interface task

Application task:

- It goes from initial robust state ('W') to failure probable state ('FP') in 10 days; $R_2 = 1/(10*24)$.
- The mean time between two consecutive failures is 3 months; $l = 1/(3*30*24)$.
- To recover from an unexpected failure it takes 30 minutes; $R1 = 2$.
- Rejuvenation is performed every 19 days; $R_4 = 1/(9*24)$.
- Time to perform rejuvenation is 10 minutes. $R_3 = 6$.

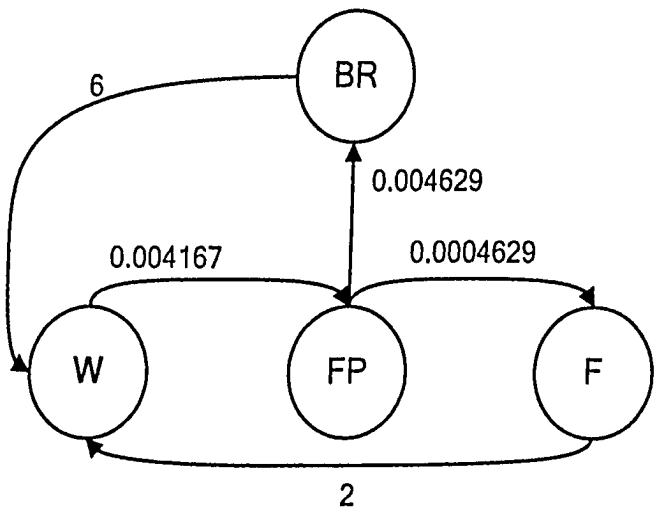

Figure 3.9 - CTMC – Application task

Database-1 task:

- It goes from initial robust state ('W') to failure probable state ('FP') in 10 days; $R_2 = 1/(10*24)$.
- The mean time between two consecutive failures is 2 months; $l = 1/(2*30*24)$.
- To recover from an unexpected failure it takes 30 minutes; $R_1 = 4$.
- Rejuvenation is performed every 19 days; $R_4 = 1/(9*24)$.
- Time to perform rejuvenation is 10 minutes. $R_3 = 6$.

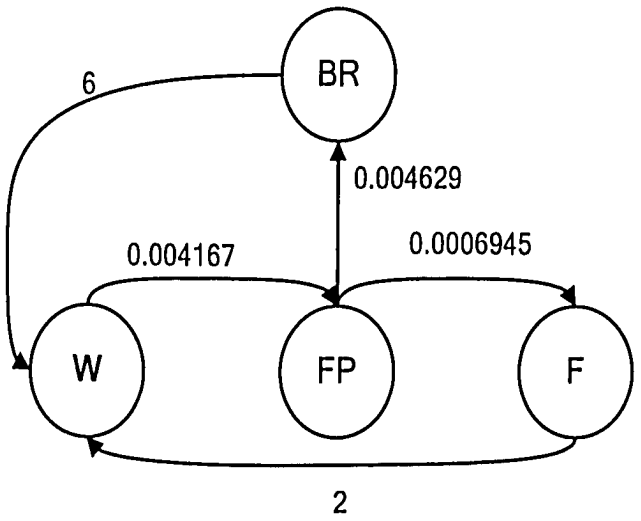

Figure 3.10 - CTMC – Database -1 task

The model parameters for Database-2 task are same as Database-1 task (as shown in figure 3.10). All the CTMCs mentioned above are solved using the SHARPE tool [47], and the steady state probability of software task residing in each of the four states is calculated. The steady state probabilities obtained considering the model parameters described above are shown in table 1. Changing the model parameters (as per the system under study) will result in change in the value of steady state probabilities. These steady state probabilities will be used further in the dependability analysis part (Step 6) of the model solution.

Table 1. Steady State Probabilities

Tasks	State F	State W	State FP	State BR
Interface	0.0002871	0.37897	0.62027	0.0004785
Application	0.0001041	0.54970	0.44985	0.0003470
Database–1	0.0001523	0.56065	0.43886	0.0003385
Database-2	0.0001523	0.56065	0.43886	0.0003385

Step 3: AND OR Graph Representation

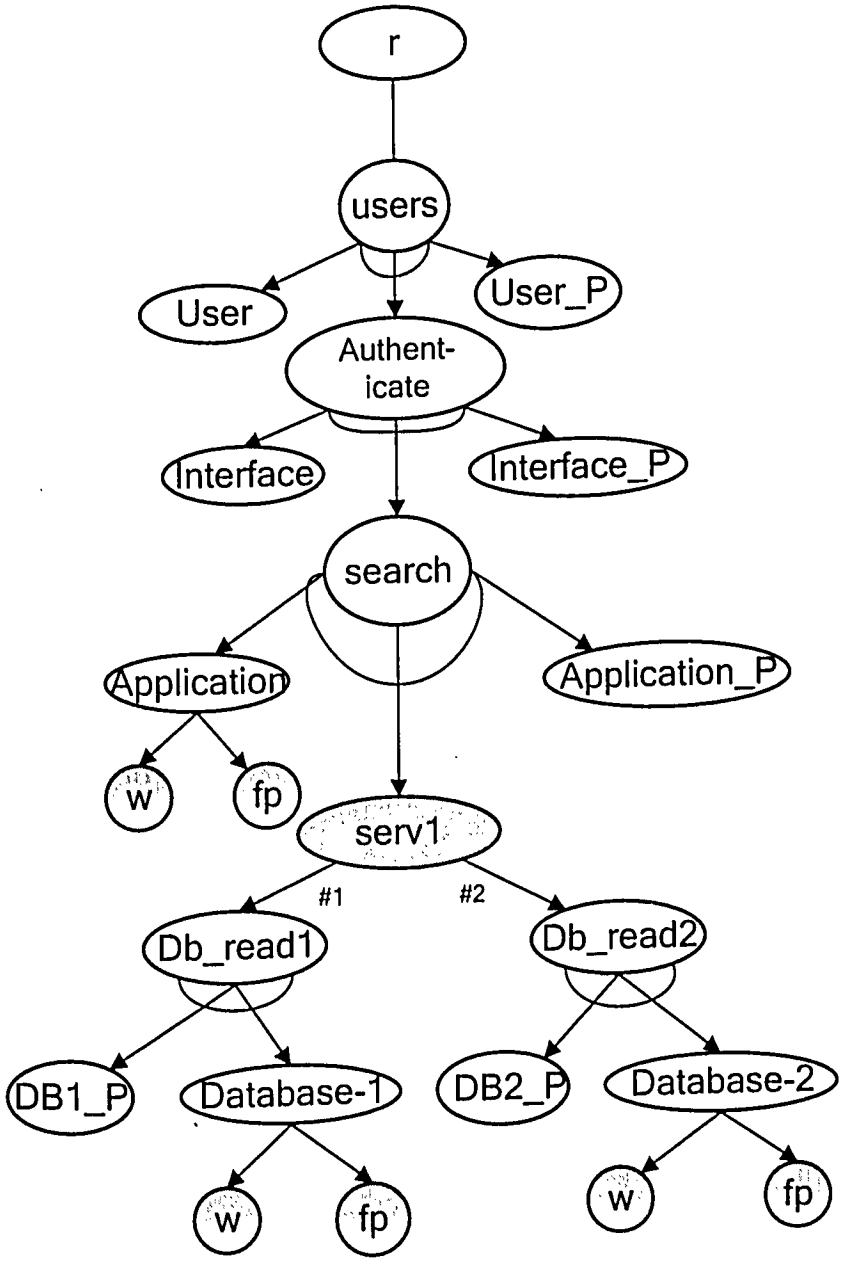

Figure 3.11 – Fault Propagation And-Or Graph

In the graph shown in figure 3.11, "users", "authenticate", "search", "read1" and "read2"are all entry nodes (AND nodes). "serv1" is the service node, which is

represented as the shaded node. "User", "Interface", "Application", "Database-1" and "Database-2" are all task nodes (OR nodes), all of which can be in any of the two different states 'W' and 'FP'. "User_P", Interface_P", "Application_P", "DB1_P" and "DB2_P" are all processor nodes (leaf nodes).

Step 4: Operational Configurations

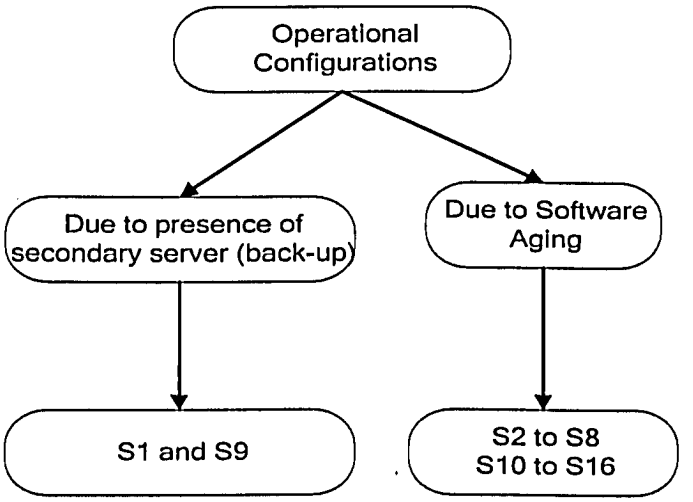

Figure 3.12 – Different Operational Configurations

S1 and S9 configurations arise due to binary states of software tasks: W and F. It does not consider the extra FP working state of tasks, as there is no consideration of software aging. For S1 we have 3 tasks and each task can be in highly robust state ('W') as well as failure probable state ('FP'). Thus we have 8 (2^3) different operational configurations including S1. E.g. *Interface-W, Application-W, Database-1-FP* is one of the possible operational configuration. Similarly for S9 also we have 8 different operational configurations including S2. E.g. *Interface-FP, Application-W, Database-1-unoperational, Database-2- FP* is one of the possible operational configuration. For all this 8 operational configurations corresponding to S9, the Database-1 task is unoperational (either failed or undergoing rejuvenation). So finally we have 16 different

operational configurations of the system as shown in table 2. The number of operational configurations is more in Rejuvenated-FTLQNS model compared to when the performance degradation due to software aging is not considered (increases from 2 to 16). Figure 3.13 and 3.14 indicates operational configuration S1 and S9 respectively.

Table 2. Operational Configurations

Notation	Operational Configurations
S1	*Interface-W, Application-W, Database-1- W*
S2	*Interface-FP, Application-W, Database-1- W*
S3	*Interface-W, Application-FP, Database-1- W*
S4	*Interface-W, Application-W, Database-1- FP*
S5	*Interface-FP, Application-FP, Database-1- W*
S6	*Interface-FP, Application-W, Database-1- FP*
S7	*Interface-W, Application-FP, Database-1- FP*
S8	*Interface-FP, Application-FP, Database-1- FP*
S9	*Interface-W, Application-W, Database-1- unoperational , Database-2- W*
S10	*Interface-FP, Application-W, , Database-1- unoperational , Database-2- W*
S11	*Interface-W, Application-FP, Database-1- unoperational, Database-2- W*
S12	*Interface-W, Application-W, Database-1- unoperational , Database-2- FP*
S13	*Interface-FP, Application-FP, Database-1- unoperational , Database-2- W*
S14	*Interface-FP, Application-W, Database-1- unoperational , Database-2- FP*
S15	*Interface-W, Application-FP, Database-1- unoperational , Database-2- FP*
S16	*Interface-FP, Application-FP, Database-1- unoperational, Database-2- FP*

Step 5: Performance Analysis (LQN)

Once we have determined the different operational configurations of the system. The next step is to generate Performance models (LQN) corresponding to every operational configuration and using LQNS tool to obtain different performance measures. Finally the desired performance measure will be assigned as a reward to every operational configuration.

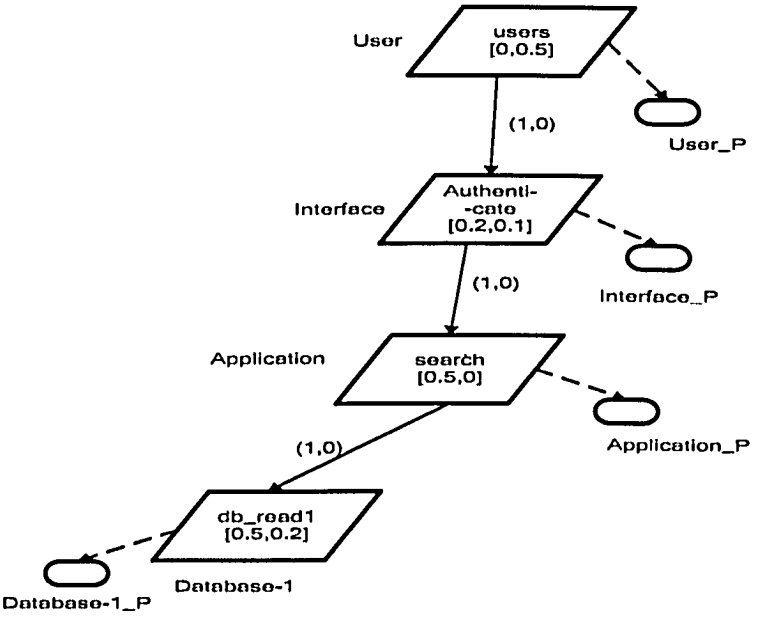

Figure 3.13 – Operational Configuration - S1

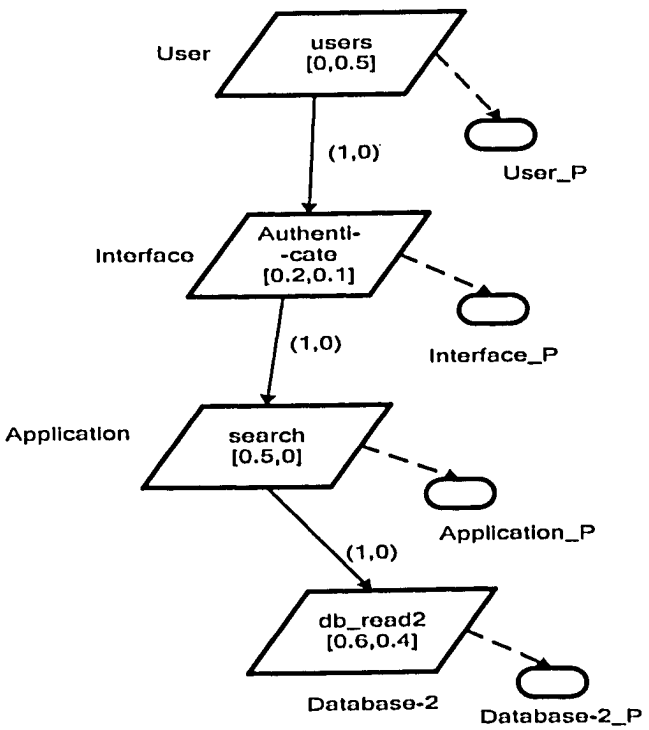

Figure 3.14 – Operational Configuration – S9

The change in the mean CPU demands for all the software tasks in "FP" state is as shown in table 3. Each and every operational configuration will have corresponding LQN model with different parameters, depending on the state of the software tasks in that particular operational configuration. All the LQN models are solved using LQNS (Layered Queueing Network Solver) tool [22]. Throughput is selected as the reward to be assigned to every operational configuration. Rewards for every operational configuration are shown in table 4. For system unoperational state, the throughput (reward) assigned is 0.

Table 3. Mean Execution Demands

Entry	Working state (W) of task	Failure Probable state (FP) of task
users	[0,0.5]	-
authenticate	[0.2, 0.1]	[0.3, 0.15]
search	[0.5, 0]	[0.75, 0]
db_read1	[0.5, 0.2]	[0.7, 0.28]
db_read2	[0.6, 0.4]	[0.84, 0.56]

Table 4. Reward rate and Probability of Operational Configurations

Operational Configurations	Reward- Throughput	Probability of occurrence
S1	0.55198	0.083811
S2	0.517833	0.137174
S3	0.490042	0.068588
S4	0.477727	0.065610
S5	0.460677	0.112257
S6	0.449909	0.107384
S7	0.429976	0.053693
S8	0.407262	0.087879
S9	0.50193	0.004230
S10	0.471498	0.006923
S11	0.450876	0.003462
S12	0.415705	0.003311
S13	0.426071	0.005666
S14	0.39498	0.005420
S15	0.381928	0.002710
S16	0.364217	0.004435
System Unoperational	0	0.24744

Step 6: Dependability Analysis (MSFT)

Apart from the reward rate for every operational configuration, we also need to determine the probability of system being in each of that configuration in order to

evaluate steady state performability. The failure probability of all the processors and reference task is assumed to be 0.05. To find the probability of operational configuration S1, following formula is used:

$Prob(S1) = (1\text{-}F\_\{Users\}) * (1\text{-}F\_\{User\_P\}) * (P\_\{Interface_w\}) * (1\text{-}F\_\{Interface\_P\}) * (P\_\{Application_w\}) * (1\text{-}F\_Application\_P\}) * (P\_\{Database\text{-}1_w\}) * (1\text{-}F\_\{Database\text{-}1\_P\})$

$= \mathbf{0.083811}$

- $F\_\{Users\}$ indicates failure probability for Users task.
- $F\_\{User\_P\}$ indicates failure probability for User processor which is used by Users task.
- $P\_\{Interface_w\}$ indicates probability of Interface task being in state 'W'. This probability is obtained by solving the CTMC for Interface task.
- $F\_\{Interface\_P\}$ indicates failure probability for Interface-Processor.
- $P\_\{Application_w\}$ indicates probability of Application task being in state 'W'. This probability is obtained by solving the CTMC for Application task.
- $F\_\{Application\_P\}$ indicates failure probability for Application-Processor.
- $P\_\{Database\text{-}1_w\}$ indicates probability of Database-1 task being in state 'W'. This probability is obtained by solving the CTMC for Database-1 task.
- $F\_\{Database\text{-}1\_P\}$ indicates failure probability for Database-1-Processor.

Similarly the probability of occurrence for operational configuration S2 to S8 can be calculated. The probabilities for operational configuration S2 to S8 are as shown in table 4. Now to calculate the probability for operational configuration S9, we first need to determine the probability of Database-1 task being unoperational. Figure 3.15 shows the MSFT which represents the combination of conditions that can cause Database-1 to be unoperational.

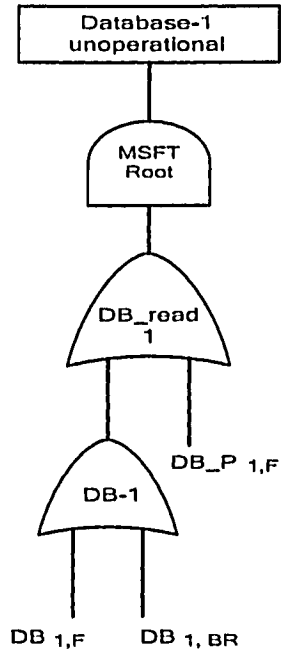

Figure 3.15 – MSFT for Database-1 unoperational

- DB $_{1,F}$ indicates that Database-1 is in 'F' state (Failed state)
- DB $_{1,BR}$ indicates that Database-1 is in 'BR' state (Being Rejuvenated state)
- DB_P $_{1,F}$ indicates that Database-1 processor is in 'F' state (Failed State)

The MSFT shown in figure 3.15 is solved using the SHARPE tool [47] to obtain the probability of Database-1 being unoperational.

To find the probability of operational configuration S9, following formula is used:

$$Prob(S9) = (1\text{-}F\_\{Users\}) * (1\text{-}F\_\{User\_P\}) * (P\_\{Interface_w\}) * (1\text{-}F\_\{Interface\_P\}) * (P\_\{Application_w\}) * (1\text{-}F\_Application\_P\}) * (P\_\{Database\text{-}2_w\}) * (1\text{-}F\_\{Database\text{-}2\_P\}) * (P\_\{Database\text{-}1\ unoperational\})$$

$$= \mathbf{0.004230}$$

- *P_{Database-1 unoperational}* indicates the probability of Database-1 task being in unoperational state obtained by solving the MSFT shown in figure 3.15.
- *P_ {Database-2_w }* indicates probability of Database-2 task being in state 'W'. This probability is obtained by solving the CTMC for Database-2 task.
- *F_ {Database-2_P}* indicates failure probability for Database-2-Processor.

Similarly we can calculate probability for all other operational configurations related to S9. Thus we will have probability of occurrence for all 16 different operational configurations as shown in table 4.

MSFT for system being in unoperational state is shown in figure 3.16. The probability obtained by solving this MSFT using SHARPE is shown in table 4. The sum of the probabilities of all 16 operational configurations and system unoperational probability adds up to 1.

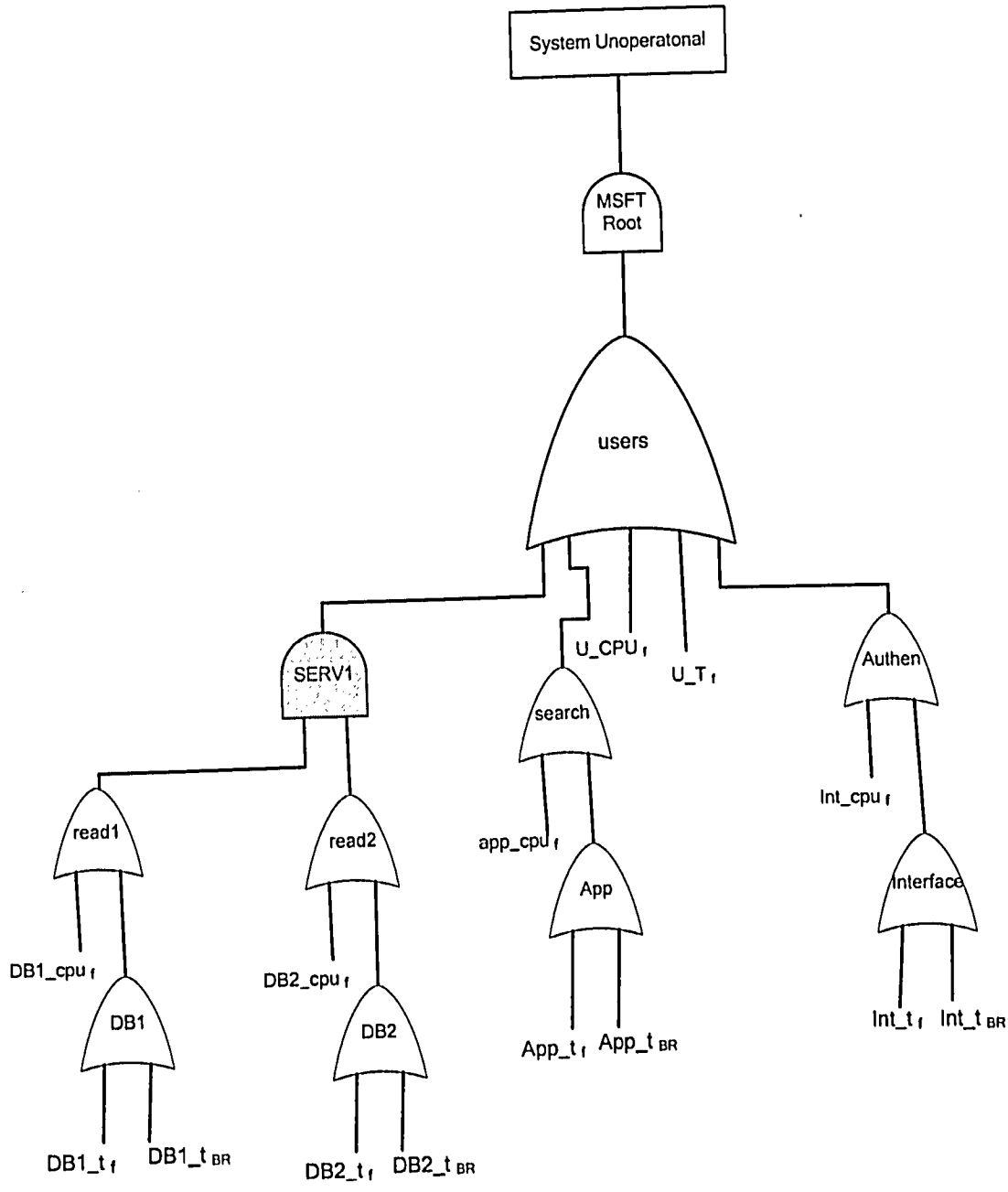

Figure 3.16 – MSFT for System unoperational

- U_CPU_f indicates the probability that processor used by Users task is in failed state.
- U_T_f indicates the probability that Users task is in failed state.
- Int_cpu_f indicates the probability that processor used by Interface task is in failed state.
- Int_t_f indicates the probability that Interface task is in failed state.

- Int_t_{BR} indicates the probability that Interface task is undergoing rejuvenation.
-App_cpu_f indicates the probability that processor used by Application task is in failed state.
- App_t_f indicates the probability that Application task is in failed state.
- App_t_{BR} indicates the probability that Application task is undergoing rejuvenation.
- DB1_cpu_f indicates the probability that processor used by Database-1 task is in failed state.
- DB1_t_f indicates the probability that Database-1 task (primary) is in failed state.
- DB1_t_{BR} indicates the probability that Database-1 task is undergoing rejuvenation.
- DB2_cpu_f indicates the probability that processor used by Database-2 task is in failed state.
- DB2_t_f indicates the probability that Database-2 task (secondary) is in failed state.
- DB2_t_{BR} indicates the probability that Database-2 task is undergoing rejuvenation.

Step 7: Steady State Performabilty Calculation

The steady state performabilty or expected steady state reward rate or mean throughput is calculated by multiplying the throughputs (rewards) associated with every operational configuration with their corresponding probabilities. In our case using the reward and probability for every operational configuration from table 4, we get the steady state performability of: **0.3569** *requests/sec.*

3.4 Summary

This chapter has introduced the Rejuvenated-FTLQN model to compute the steady state performability of the system under the effects of software aging and rejuvenation. CTMCs were used for modeling aging and rejuvenation, where the degradation was described by a two step process. Fault propagation And-Or graph was used to generate different operational configurations of the system, LQN model corresponding to every configuration was evaluated to obtain the throughput and MSFT was used in the dependability analysis part of the calculations. A simple example of customer-information retrieval system was used to demonstrate all the steps involved in the Rejuvenated-FTLQN model solution.

Chapter 4
Analysis using Rejuvenated-FTLQN Model

This chapter discusses about following things. Section 4.1 deals with the impact of performing rejuvenation on the steady state performability of the system. The Rejuvenated-FTLQN model solution is compared for two cases: (1) With Rejuvenation and (2) Without Rejuvenation. Section 4.2 discusses about the effects of increasing the rejuvenation frequency on the steady state performability. Section 4.3 compares the Rejuvenated-FTLQN model solution for two different cases: (1) system with fault tolerance (2) system without fault tolerance. For both the cases the effects of time to perform rejuvenation, rejuvenation frequency, and base longevity interval on steady state performability is studied. Section 4.4 deals with the comparison of different designs of the system based on steady state performability.

4.1 Model Solution with and without Rejuvenation

In this section the example of FTLQN model (figure 3.7) given in chapter 3 is used. The Rejuvenated-FTLQN model solution explained in chapter 3 was taking into consideration that rejuvenation was implemented. In this section we will compare that model solution with one in which no rejuvenation is performed.

The CTMC for the software task when no rejuvenation is performed is as shown in figure 4.1, as explained in section 3.2 of chapter 3. The CTMC for the software task when rejuvenation is performed is shown in figure 4.2. Each of the four states is described as follows:

- **State W**: Working or highly robust state (normal operation)
- **State FP**: Failure Probable state (due to aging).
- **State BR**: Being Rejuvenated state (undergoing rejuvenation)
- **State F**: Failed State.

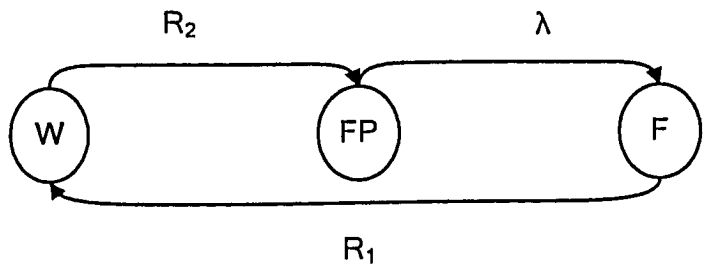

Figure 4.1 - CTMC – Without Rejuvenation

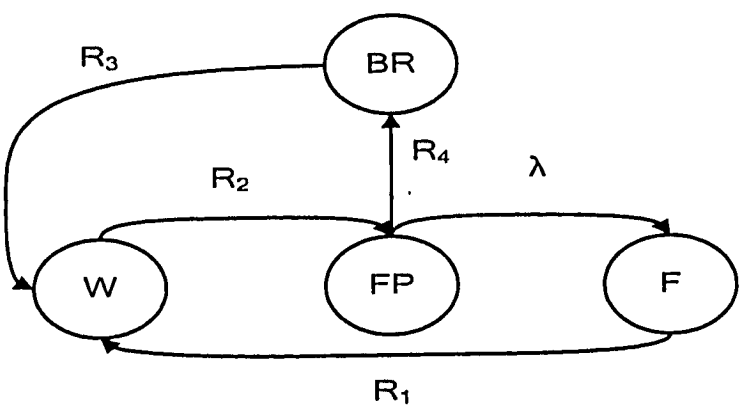

Figure 4.2 - CTMC – With Rejuvenation

The rates in the CTMCs for every software tasks are same as described in chapter 3. All the rates in the CTMC for different software tasks are shown in table 5. The steady state probabilities obtained after solving the CTMCs using SHARPE tool is shown in table 6 and 7. Table 6 shows the probabilities for "Without Rejuvenation" case and table 7 shows the probabilities for "With Rejuvenation" case.

Table 5. Rates in CTMCs

Tasks	Rate R_1	Rate R_2	Rate R_3	Rate R_4	Rate λ
Interface	1	0.008334	6	0.0046296	0.000462
Application	2	0.004167	6	0.0046296	0.000462
Database–1	4	0.004167	6	0.0046296	0.000694
Database-2	4	0.004167	6	0.0046296	0.000694

Table 6. Steady state probabilities- Without Rejuvenation

Tasks	State F	State W	State FP
Interface	0.000438	0.052598	0.94696
Application	0.000208	0.099960	0.899383
Database–1	0.000297	0.142810	0.856890
Database-2	0.000297	0.142810	0.856890

Table 7. Steady state probabilities- With Rejuvenation

Tasks	State F	State W	State FP	State BR
Interface	0.0002871	0.37897	0.62027	0.0004785
Application	0.0001041	0.54970	0.44985	0.0003470
Database–1	0.0001523	0.56065	0.43886	0.0003385
Database-2	0.0001523	0.56065	0.43886	0.0003385

The fault propagation AND-OR graph is same for both the cases as shown in figure 3.7. We will have same number of operational configurations as before (16). The rewards (throughput) associated with the operational configurations will also remain the same. The operational configurations (their notation) and the throughputs are shown in table 8.

The difference is in the configuration probabilities. When no rejuvenation is performed the software task will reside in failure probable state ('FP') for longer duration compared to when rejuvenation is performed. For example, say operational configuration- **S8** - *Interface-FP, Application-FP, Database-1- FP* from table 8, we can see that when rejuvenation is implemented the system stays in this configuration for less amount of time compared to when rejuvenation is not implemented. The configuration probabilities for all the operational configurations are shown in table 8.

Table 8. Reward rate and Probability of Operational Configurations

Operational Configurations	Reward- Throughput	Probability -*With Rejuvenation*	Probability- *Without Rejuvenation*
S1	0.55198	0.083811	0.000581
S2	0.517833	0.137174	0.01046
S3	0.490042	0.068588	0.005227
S4	0.477727	0.065610	0.003486
S5	0.460677	0.112257	0.094114
S6	0.449909	0.107384	0.062763
S7	0.429976	0.053693	0.031366
S8	0.407262	0.087879	0.564701
S9	0.50193	0.004230	0.000029
S10	0.471498	0.006923	0.000526
S11	0.450876	0.003462	0.000263
S12	0.415705	0.003311	0.000175
S13	0.426071	0.005666	0.004732
S14	0.39498	0.005420	0.003156
S15	0.381928	0.002710	0.001577
S16	0.364217	0.004435	0.028393

- **Without Rejuvenation:**

  Steady state performability=**0.3306** *requests/*sec.

- **With Rejuvenation:**

  Steady state performability=**0.3569** *requests/*sec.

The result shows that the steady state performability increases when rejuvenation is performed.

4.2 Effects of Rejuvenation Rate

This section describes the effects of rejuvenation rate (frequency) on the steady state performability of the system. Rejuvenation rate refers to the rate R_4 in the CTMC shown in figure 4.2, and rate R_3 indicates the repair rate after a rejuvenation event. As we increase the rejuvenation rate, the steady state probabilities of task being in each of the four states ('W', 'FP', 'BR', 'F') changes. The steady state probability of task being in highly robust state ('W') increases. The result is that the probability of occurrence of the operational configurations changes along with the inputs for MSFT. Finally it results into changed steady state performability value.

Rejuvenation rate for database servers and application task is increased. On the x-axis of the graph (figure 4.3) only rejuvenation rate for Application task is shown but the plot takes into consideration increase in rejuvenation rate for database servers too. All other model parameters are same as described in table 5. Initially when we increase the rejuvenation frequency, we see increase in steady state performability. The increase in the value of steady state performability is only upto a point (0.38), after which it starts declining. For e.g. increasing the rejuvenation rate from 0.0046 to 0.0069 results in increase in steady state performability from 0.35 to 0.36. Steady state performability decreases from 0.38 to 0.375 when the rejuvenation rate in increased from 0.02 to 0.04, as the system which is rejuvenated very often might also lose availability.

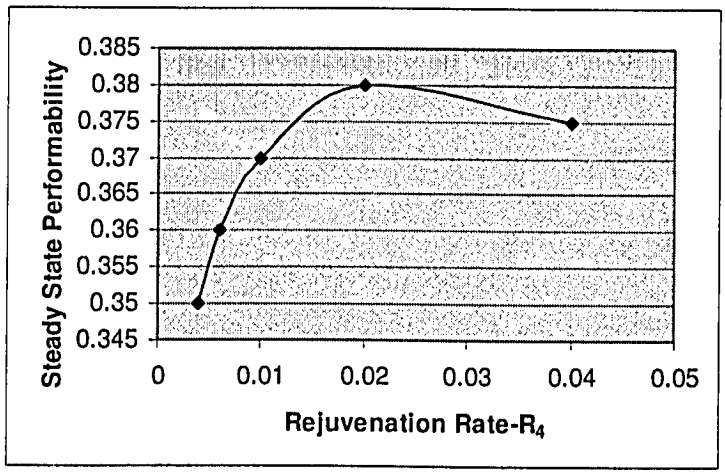

Figure 4.3 - Steady State Performability (SSP) v/s Rejuvenation Rate

4.3 Effects of Fault Tolerance

This section describes the effects of fault tolerance on the steady state performability. Rejuvenated-FTLQN model solution is applied to the system without fault tolerance and with fault tolerance. Also the effects of rejuvenation frequency, time to perform rejuvenation and base longevity interval on the steady state performability is compared for two different designs:

(1) System with no fault tolerance- shown in figure 4.4.
(2) System with fault-tolerance- shown in figure 4.5.

In case (1) we don't have secondary database server available. So when the primary database server is unoperational (Undergoing Rejuvenation or Failed) the system is unoperational. But in case (2) when primary database server is unoperational, the system is still operational with Application task using secondary database. In case (1) we have 8 different operational configurations (S1 to S8), considering the fact that the software tasks have 2 operational states ('W' and 'FP'). In case (2) we have 16 different operational configurations (S1 to S16) as explained in chapter 3.

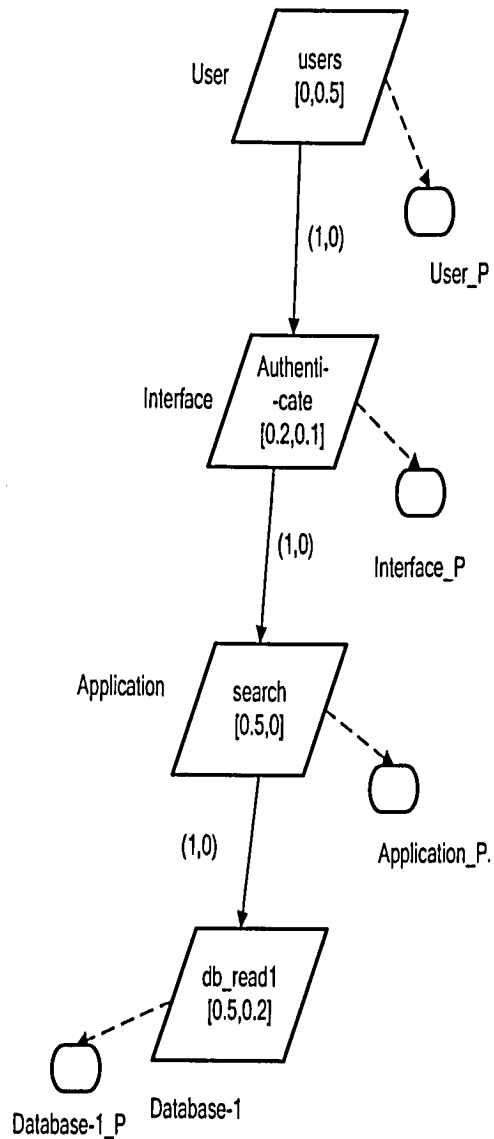

Figure 4.4 –No fault tolerance

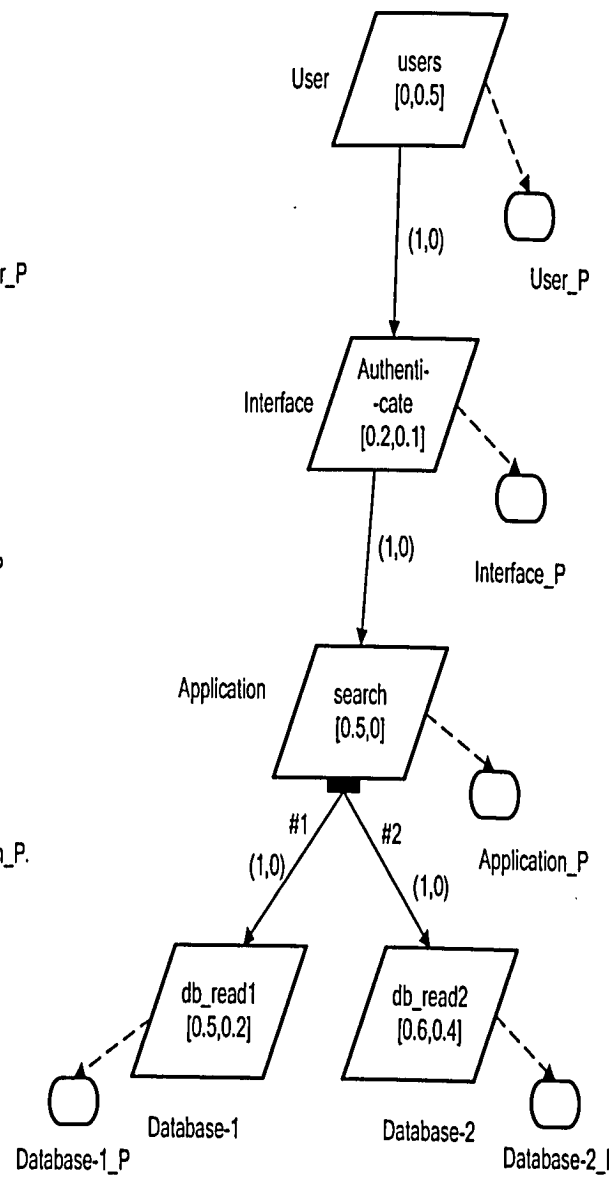

Figure 4.5 – With fault tolerance

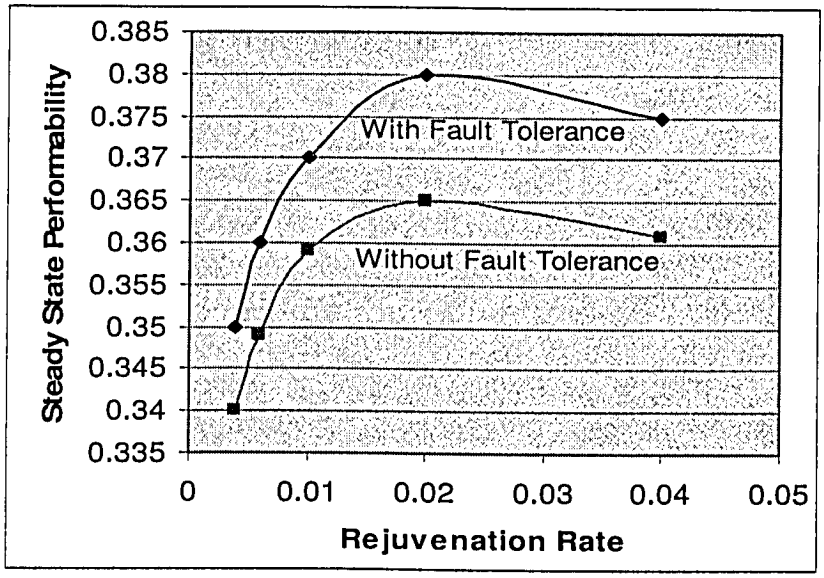

Figure 4.6 - SSP v/s Rejuvenation Rate- With and Without Fault Tolerance

The rejuvenation rate for database servers and Application task is increased. On the x-axis of the graph (figure 4.6) only rejuvenation rate for Application task is shown but the plot takes into consideration increase in rejuvenation rate for database servers too. All other model parameters are same as described in table 5. From figure 4.6, we can see that steady state performability is higher when we have secondary database available. Initially when we increase the rejuvenation frequency we see increase in steady state performability. But after certain threshold (e.g. 0.365 for system without fault tolerance) the steady state performability starts declining. Thus the increase in the value of steady state performability is only upto a point, after which it starts decreasing. "Rejuvenated-FTLQN" model can be used to study the effects of rejuvenation frequency on the steady state performability of the system and also to determine the threshold after which it has a negative impact.

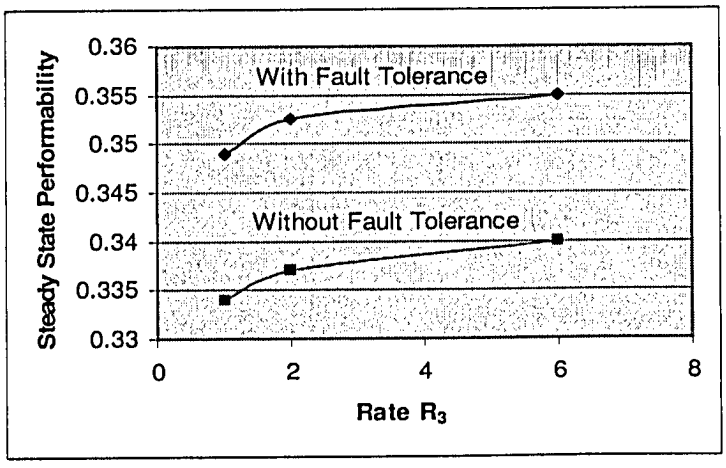

Figure 4.7 - SSP v/s Rate R_3- With and Without Fault Tolerance

If we increase rate R_3 and keep R_4 constant then the steady state performability increases as shown in figure 4.7. On the x-axis of the graph (figure 4.7) only Rate R_3 for Application task is shown but the plot takes into consideration increase in rates for database servers too. When we increase rate R_3, the software tasks stays in "Being Rejuvenated" state ('BR') for shorter duration and in highly robust state ('W') for longer duration, resulting in higher steady state performability.

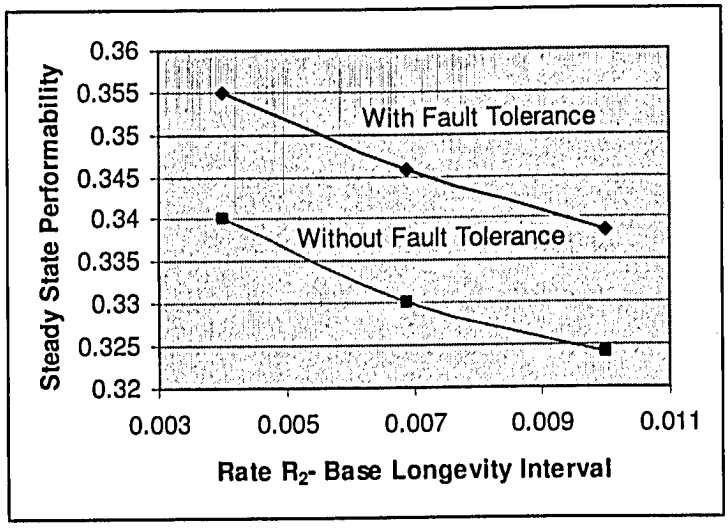

Figure 4.8 - SSP v/s Rate R_2- With and Without Fault Tolerance

If we increase the rate at which the application goes from 'W' to 'FP' state (rate R_2), and keep all other model parameters same then the steady state performability decreases

as shown in figure 4.8. This is because when we increase the rate R_2, the software tasks stays in highly robust state for shorter duration and in failure probable state for longer duration (performing at degraded performance level for longer amount of time) which results in lower steady state performability value. Thus Rejuvenated-FTLQN model can also be used to analyze the effects of time taken to perform rejuvenation and base longevity interval on the steady state performability.

4.4 Comparing Different Designs

In this section, three different designs of the system introduced in chapter 3 are compared. Rejuvenated-FTLQN model is used to evaluate the steady state performability under the effects of software aging and rejuvenation for all the three designs. The main purpose is to illustrate how Rejuvenated-FTLQN model can be used to compare different designs of the system based on the steady state performability.

- **Design 1**: Fault tolerance at Database tier- figure 4.9.
- **Design 2**: Fault tolerance at Application tier- figure 4.10.
- **Design 3**: Fault tolerance at Web tier- figure 4.11.

The secondary tasks have the same failure profiles as primary tasks. We will have 16 different operational configurations for each of the design, but the difference is in the layer in which the fault tolerance is available. Two operational configurations that arise due to presence of secondary server alone and not due to software aging, for design 1, design 2 and design 3 are shown in table 9, table 10, and table 11 respectively.

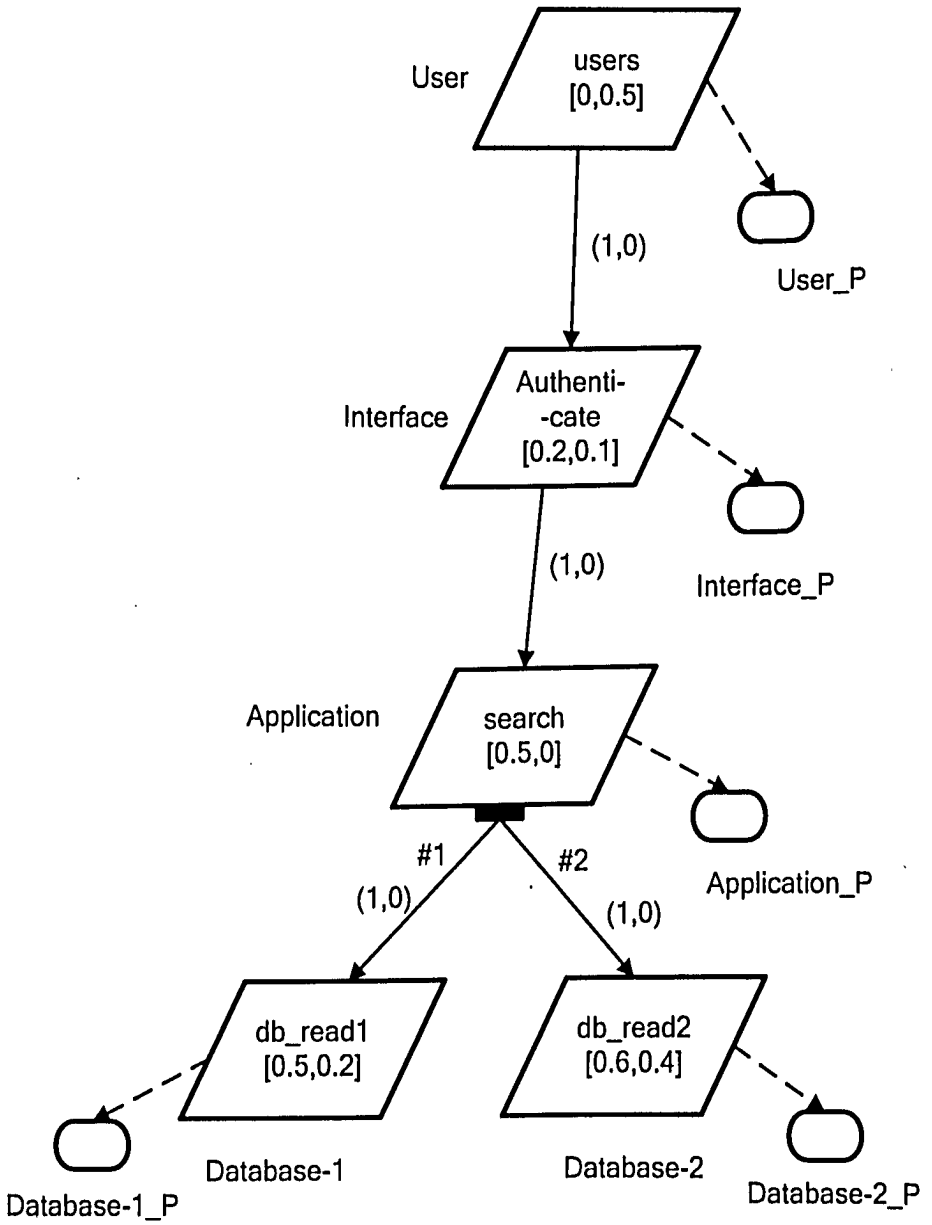

Figure 4.9 - Design 1 - FTLQN Model

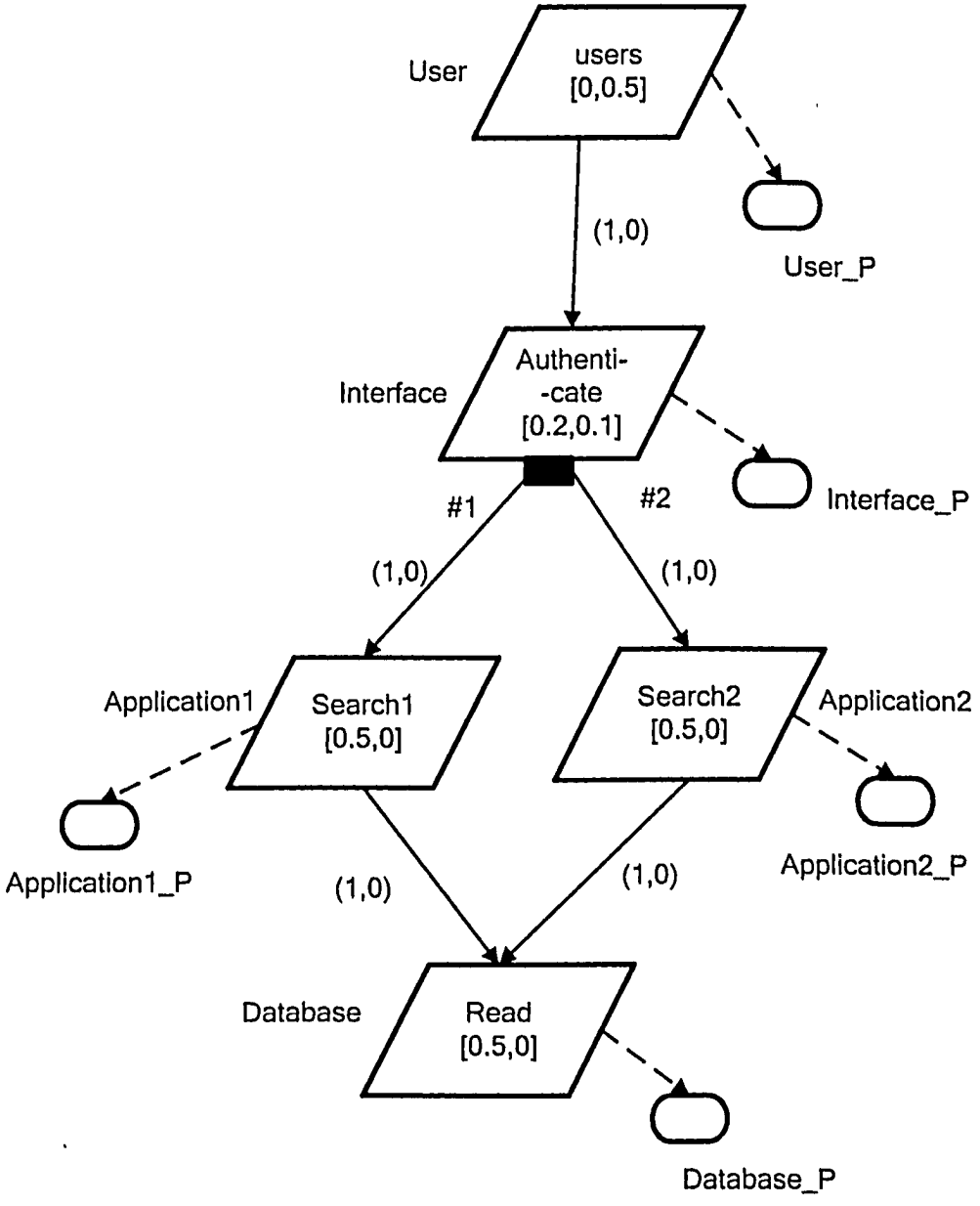

Figure 4.10 - Design 2 - FTLQN Model

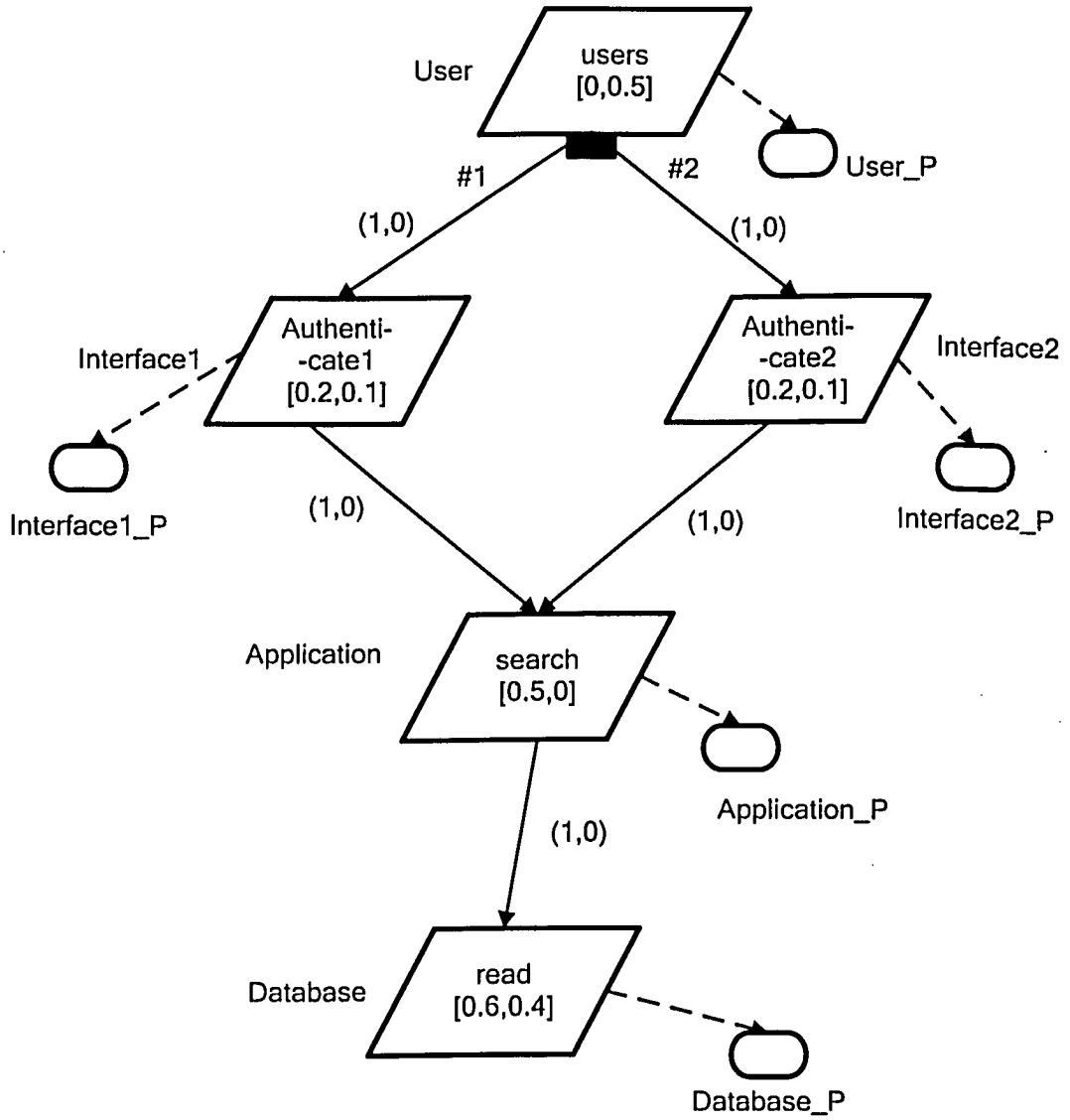

Figure 4.11 - Design 3 - FTLQN Model

Table 9. Design 1- Operational Configurations (due to presence of secondary server only)

Notation	Operational Configurations
D1OC1	*Interface-W, Application-W, Database-1- W*
D1OC2	*Interface-FP, Application-FP, Database-1- unoperational, Database-2- W*

Table 10. Design 2- Operational Configurations (due to presence of secondary server only)

Notation	Operational Configurations
D2OC1	*Interface-W, Application-1-W, Database-W*
D2OC2	*Interface-W, Application-1-unoperational, Application-2-W Database-1- W*

Table 11. Design 3- Operational Configurations (due to presence of secondary server only)

Notation	Operational Configurations
D3OC1	*Interface-1-W, Application-W, Database-W*
D3OC2	*Interface-1-unoperational, Interface-2-W, Application-W, Database-W*

The mean execution demands (or mean CPU demands) for primary as well as secondary entries of the tasks are shown in table 12. In all the three cases, secondary is less powerful compared to primary. All the three FTLQN models are solved using the Rejuvenated-FTLQNS tool, and the steady state performability values obtained for each of them is shown in the table 13.

Entry	Primary- Mean CPU demands	Secondary- Mean CPU demands
read	[0.5, 0.2]	[0.6, 0.4]
search	[0.5, 0]	[0.7, 0]
authenticate	[0.2, 0.1]	[0.3, 0.2]

Table 12. Mean CPU demands for Primary and Secondary

Table 13. Steady State Performability for three different designs

Design	Steady State Perfomability
Design 1	0.3569 *requests*/sec
Design 2	0.3646 *requests*/sec
Design 3	0.3648 *requests*/sec

From table 13, we can see that the steady state performability is almost same for design 2 and design 3 of the system, which is little higher compared to design 1. Thus Rejuvenated-FTLQN model can be used to evaluate different proposed designs of the system and the results can be compared based on steady state performability values. The results obtained can also help in making decision of whether adding extra hardware will be useful and by how much. Trade-off between the performability value and the cost of extra hardware can be evaluated to make a decision. For e.g. from table 3 we can see that adding extra server at application tier or database tier results in approximately same value of steady state performability. Thus the decision of buying the server with lower cost can be made.

4.5 Summary

In this chapter, the impact of performing rejuvenation on steady state performability was studied using the Rejuvenated-FTLQN model. The effect of changing rejuvenation rate on steady state performability was considered. The gain in the value of steady state performability due to fault tolerance in the system and the effects of time to perform rejuvenation, base longevity interval and rejuvenation frequency on system with and

without fault tolerance were considered. Finally, Rejuvenated-FTLQN model was used to compare different designs of the system based on steady state performability.

Chapter 5

Rejuvenated-FTLQN Model Solver Description

This chapter presents a high level description of the Rejuvenated- FTLQNS (Fault Tolerant Layered Queueing Network Solver) tool, which automates the Rejuvenated-FTLQN Model solution. Section 5.1 gives an overview of Rejuvenated-FTLQNS tool. Section 5.2 describes the processing steps that are carried out by Rejuvenated-FTLQNS tool with the help of the block diagram. Section 5.3 describes how to use the Rejuvenated-FTLQNS tool through command line.

5.1 Overview of Rejuvenated-FTLQNS tool

This section gives a brief overview of Rejuvenated-FTLQNS tool. This tool automates the evaluation of performability for fault tolerant systems under the effects of software aging and rejuvenation.

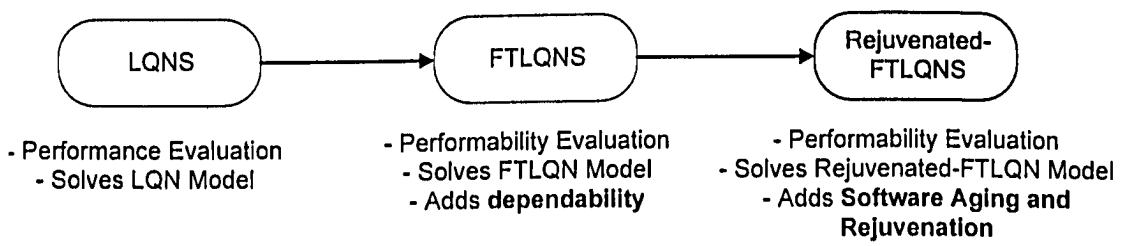

Figure 5.1 - LQNS v/s FTLQNS v/s Rejuvenated-FTLQNS

LQNS (Layered Queueing Network Solver) is the software tool used for performance evaluation [60]. LQNS solves LQN model. In the Rejuvenated-FTLQN model solution we are generating LQN models (performance models) for ever operational configuration

and LQNS is invoked to solve the LQN models. FTLQNS (Fault Tolerant Layered Queueing Network Solver) is the software tool that was developed for performability evaluation [10]. FTLQNS adds dependability evaluation to the existing LQNS tool and automates the algorithm to evaluate the steady state performability of fault tolerant layered distributed systems. In the evaluation of performability by FTLQNS tool there is no consideration of software aging and rejuvenation. So Rejuvenated-FTLQNS tool which was developed in present research adds the effects of software aging and rejuvenation to the FTLQNS performability computations. Detailed architecture of the tool is discussed in next section of this chapter.

5.2 Description of Rejuvenated-FTLQNS tool

A high level block diagram of the Rejuvenated-FTLQNS tool is shown in figure 5.2. A Rejuvenated-FTLQN model can be described in a plain text file. For example, the input file for the model shown in figure 5.2 is shown in figure 5.3. The Rejuvenated-FTLQNS Parser takes a Rejuvenated-FTLQN model description that obeys the BNF grammar given in Appendix A and develops the corresponding *fault-propagation AND-OR graph*. For every task except reference task, the rates in the CTMC are mentioned in the task description section of the input file. The rates are specified in this order: R_1, R_2, R_3, R_4, and λ. For the input file shown in figure 5.3, the rates are shown in bold font.

The AND-OR Graph Analyzer is a C++ routine that operates on the *fault-propagation AND-OR graph* and generates the operational configurations due to fault tolerance. This part is same as the FTLQNS tool. From the knowledge of the input model and the operational configurations generated due to presence of secondary server, another C++ routine generates all the operational configurations due to software aging. It takes into account the multiple operational states (W and FP) for the software tasks. For the example shown in figure 5.2, we will have 8 different operational configurations. 8 LQN input files will be generated by Rejuvenated-FTLQNS tool and LQNS tool will be iteratively invoked to solve the LQN models. One of the LQN input file is given in

Appendix B. LQNS tool will generate 8 output files corresponding to every LQN input file. A PERL script extracts the desired value (throughput) from the output file. This throughput value is assigned as the reward to the corresponding operational configuration. A sample output file generated by LQNS tool and the PERL script that extracts the throughput value from it are shown in Appendix B.

The AND-OR Graph Analyzer along with the model description generates the MSFTs. The sample MSFT input file for the example shown in figure 5.2 is given in Appendix B. The SHARPE tool is then invoked to solve the MSFTs. A PERL script extracts the desired value (failure probability) from the output file generated by SHARPE tool. These failure probabilities are then used to calculate the probabilities of occurrence for operational configurations. A sample output file generated by SHARPE tool and the PERL script that extracts the failure probability from it are shown in Appendix B.

Finally, the probabilities of the operational configurations and the rewards (throughput) are fed to the Performability Calculator to compute the steady-state performability. The output of the Rejuvenated-FTLQNS tool consists of the different operational configurations, their probabilities, their associated reward rates and the steady state performability.

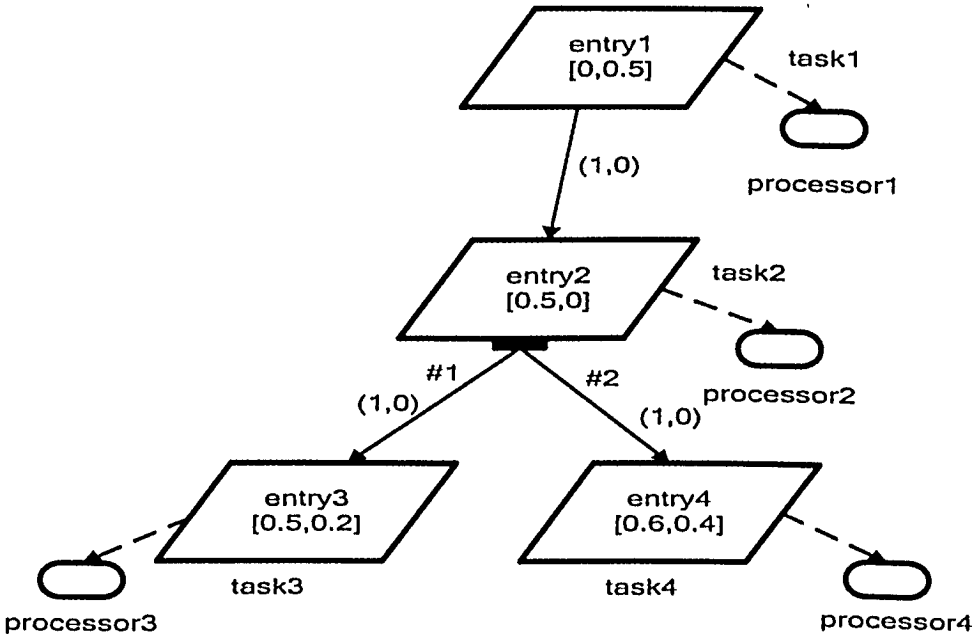

Figure 5.2 - FTLQN Model

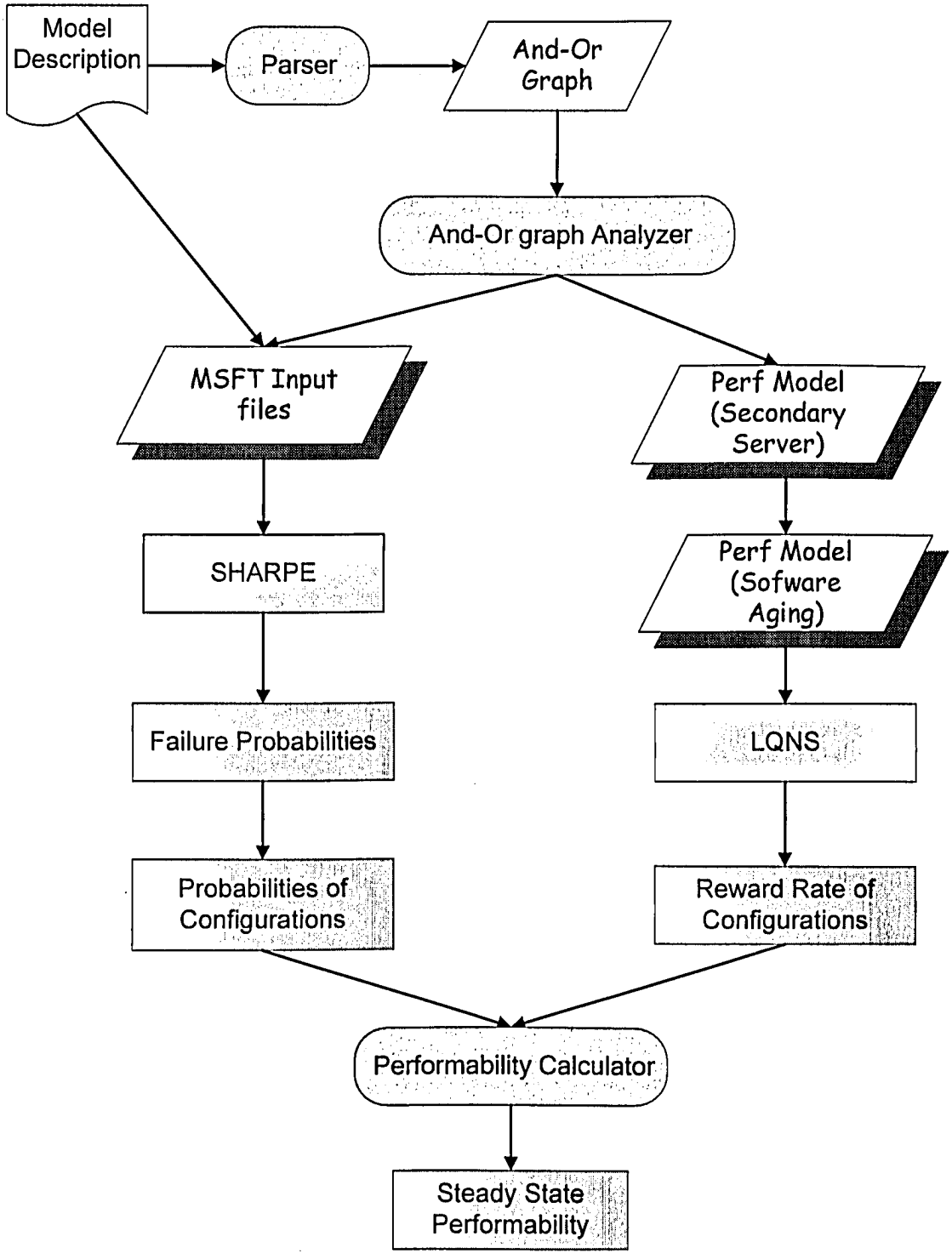

Figure 5.3 - Rejuvenated-FTLQNS tool – Block Diagram

```
G "A1" 1e-06 50 5 0.900000 -1
P 4
p p1 f (0.05)
p p2 f (0.07)
p p3 f (0.05)
p p4 f (0.05)
-1

T 4
t t1 r e1 -1 p1 (0.1)
t t2 n e2 -1 p2 (1)( 0.008334) (6) (0.0046296) (0.000462)
t t3 n e3 -1 p3 (2)(0.002976) (3) (0.002976) (0.0001157)
t t4 n e4 -1 p4 (2)(0.002976) (3) (0.002976) (0.0001157)
-1

E 4
s e1   0.000000  0.500000   -1
s e2   0.500000  0.000000   -1
s e3   0.500000  0.200000   -1
s e4   0.600000  0.400000   -1
y e1 e2 1 0 -1
ALT y e2 [e3, e4] 1.0  0.0   -1
-1
```

Figure 5.4 - Rejuvenated-FTLQNS Input File

5.3 How to use Rejuvenated-FTLQNS tool

This section describes how to use Rejuvenated-FTLQNS tool through command line. The Rejuvenated-FTLQNS tool is invoked as follows:

% Rejuvenatedftlqns *filename*

The *filename* should have *.ftlqn* extension, for example, *ex1.ftlqn*. The description in the file should follow the BNF format given in Appendix A.

For the FTLQN model shown in figure 5.2, the input file is shown in figure 5.3. In Figure 5.3, first section describes the four processors, second section describes all the tasks along with the associated CTMC rates and the last section describes the CPU demand for the four entries along with the service dependencies among the entries. Assuming the name of the input file to be example.ftlqn, the Rejuvenated-FTLQNS is invoked on this file as follows:

% Rejuvenatedftlqns example.ftlqn

The sample intermediate outputs generated by Rejuvenated-FTLQNS tool are shown in Appendix B. The final output generated is shown in Figure 5.5 below:

```
Steady State Performability of the System: 0.433324 requests/sec
```

Figure 5.5 – SSP calculated by Rejuvenated-FTLQNS

5.4 Summary

In this chapter, a high level description of Rejuvenated-FTLQNS tool which automates the Rejuvenated-FTLQN model solution was given. This chapter also described the processing steps that are carried out by the Rejuvenated-FTLQNS tool.

Chapter 6
Case Study: Building Security System (BSS)

A building security system is a complex distributed system which can be used at hospitals, hotels, and laboratories etc [63]. This chapter uses the "Rejuvenated-FTLQN" model to evaluate the steady state performability of the Building Security System (BSS), taking into consideration the impact of software aging and rejuvenation. First the overview and the usage of the system is given. Then the LQN, FTLQN and "Rejuvenated-FTLQN" model of the system are discussed. Finally the effects of changing rejuvenation rate, base longevity interval and time to perform rejuvenation on steady state performability are also considered. The main purpose of this chapter is to show that Rejuvenated-FTLQN model can be used to analyze a large distributed system involving many software tasks, with the help of an example (BSS).

6.1 Description of Building Security System (BSS)

This section gives a brief overview of Building Security System. This system is mainly used for following two purposes:

- To control access to a building (*Access Control Scenario*).
- To monitor activity in a building (*Acquire/Store Video Scenario*).

Apart from the two main scenarios stated above, the system can be used for following purposes also:

- Operations for administration of the access rights.
- Viewing the video frames.

- Sending an alarm after multiple access failures.

6.1.1 Description of Two Main Scenarios

This section explains the *Access Control Scenario* and *Acquire/Store Video Scenario.*

- **Access Control Scenario**

  In the Access Control scenario following steps takes place:

  1. A card is inserted into a door-side reader.
  2. A door-side reader reads and transmits the data to a server.
  3. Server checks the access rights associated with the card in a database of access rights.
  4. Then either triggers the lock to open the door or denies access.

- **Acquire/Store Video Scenario**

  In the Acquire/Store Video Scenario following steps takes place:

  1. Number of web cameras located around the building.
  2. Video frames are captured periodically by these web cameras.
  3. This Video frames are stored in the database.

6.1.2 Main Components of Building Security System

Figure 6.1 shows the main components involved in the Building Security System.

- **Application Server:** To control access to the building (AccessController) and to process images captured and store it into the database.
- **Security Card Reader:** To read the information from the card.
- **Video Camera:** To Capture the video frames.
- **Door Lock Actuator:** Controls the door lock.
- **Database Server:** Store the access rights as well as the images that are captured by the video cameras.

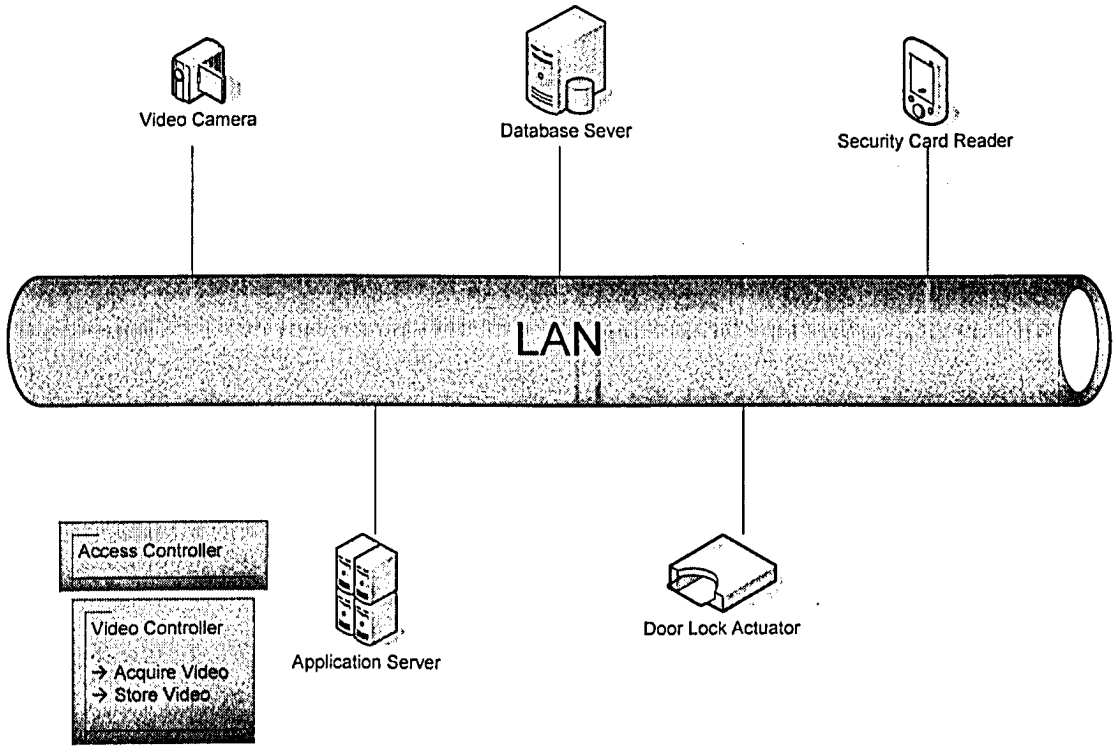

Figure 6.1 - Building Security System- Main Components

6.2 Scenarios for Building Security System (BSS)

Unified Modeling Language (UML) is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system, referred to as a UML model [5]. In this section, a UML sequence diagram is given for access control scenario and acquire/store video scenario [63].

- **UML Sequence diagram**

A type of interaction diagram, a *sequence diagram* shows the actors or objects participating in an interaction and the events they generate arranged in a time sequence.

The sequence diagram is used primarily to show the interactions between objects in the sequential order that those interactions occur. The focus is less on messages themselves and more on the order in which messages occur; nevertheless, most sequence diagrams will communicate what messages are sent between a system's objects as well as the order in which they occur. The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to .

Thus, the vertical dimension in a sequence diagram represents time, with time proceeding down the page. The horizontal dimension represents different actors or objects.
The UML scenario diagram for access control scenario and acquire/store video scenario are as shown below:

- **Access Control Scenario – UML Sequence Diagram**

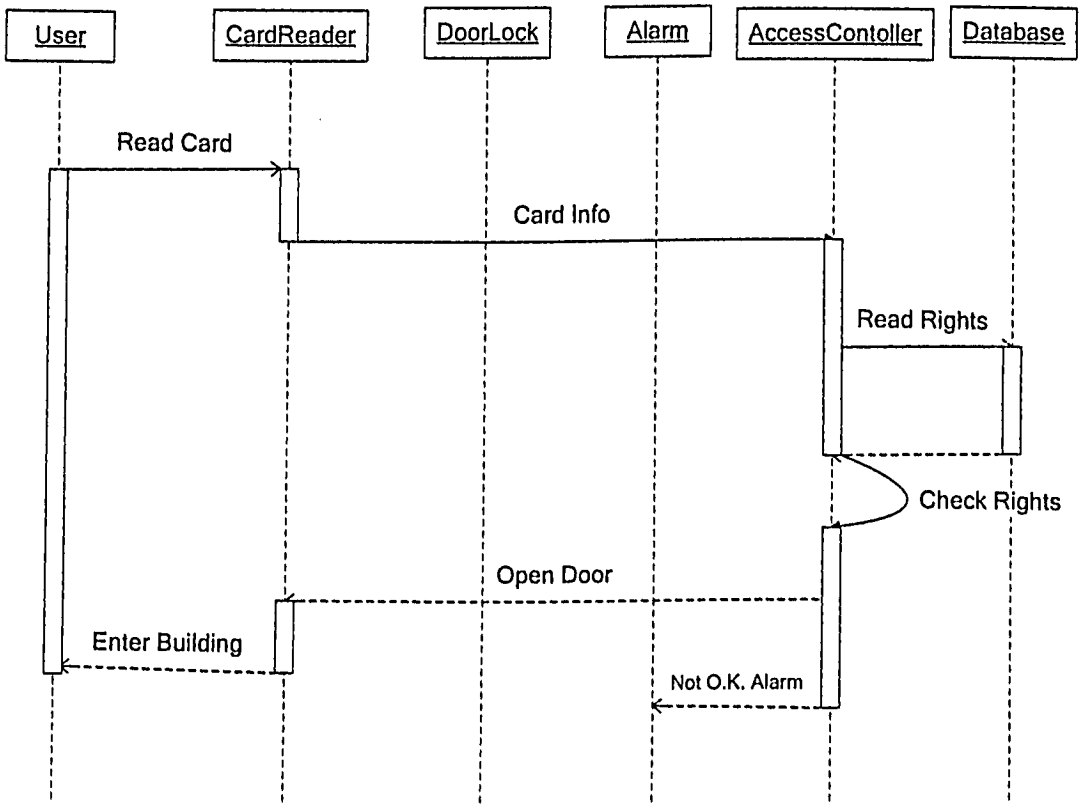

Figure 6.2 - Sequence Diagram for Access Control Scenario

- **Acquire/Store Video Scenario –UML Sequence Diagram**

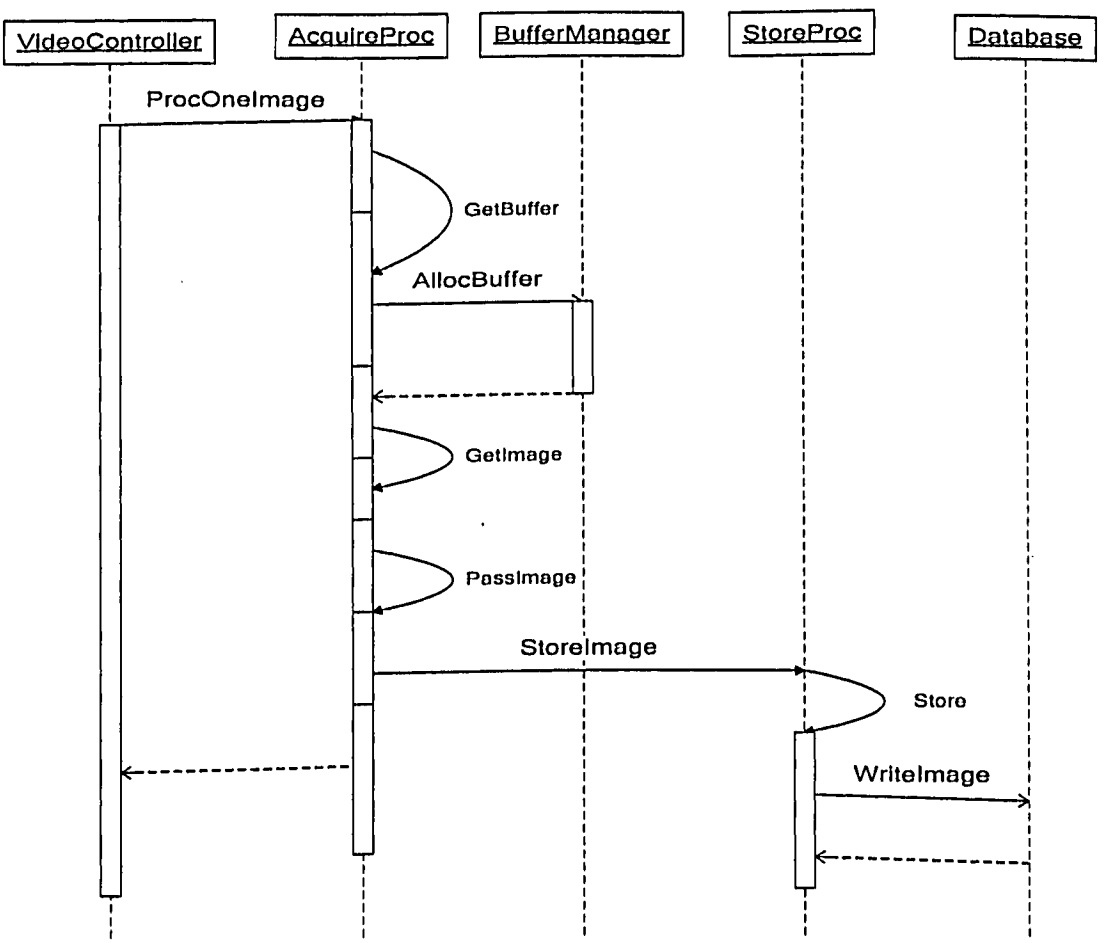

Figure 6.3 - Sequence Diagram for Acquire/Store Video Scenario

6.3 LQN and FTLQN Model of Building Security System

This section describes the LQN and FTLQN model of the Building Security System. The LQN Model of the system is shown in figure 6.4 [63].

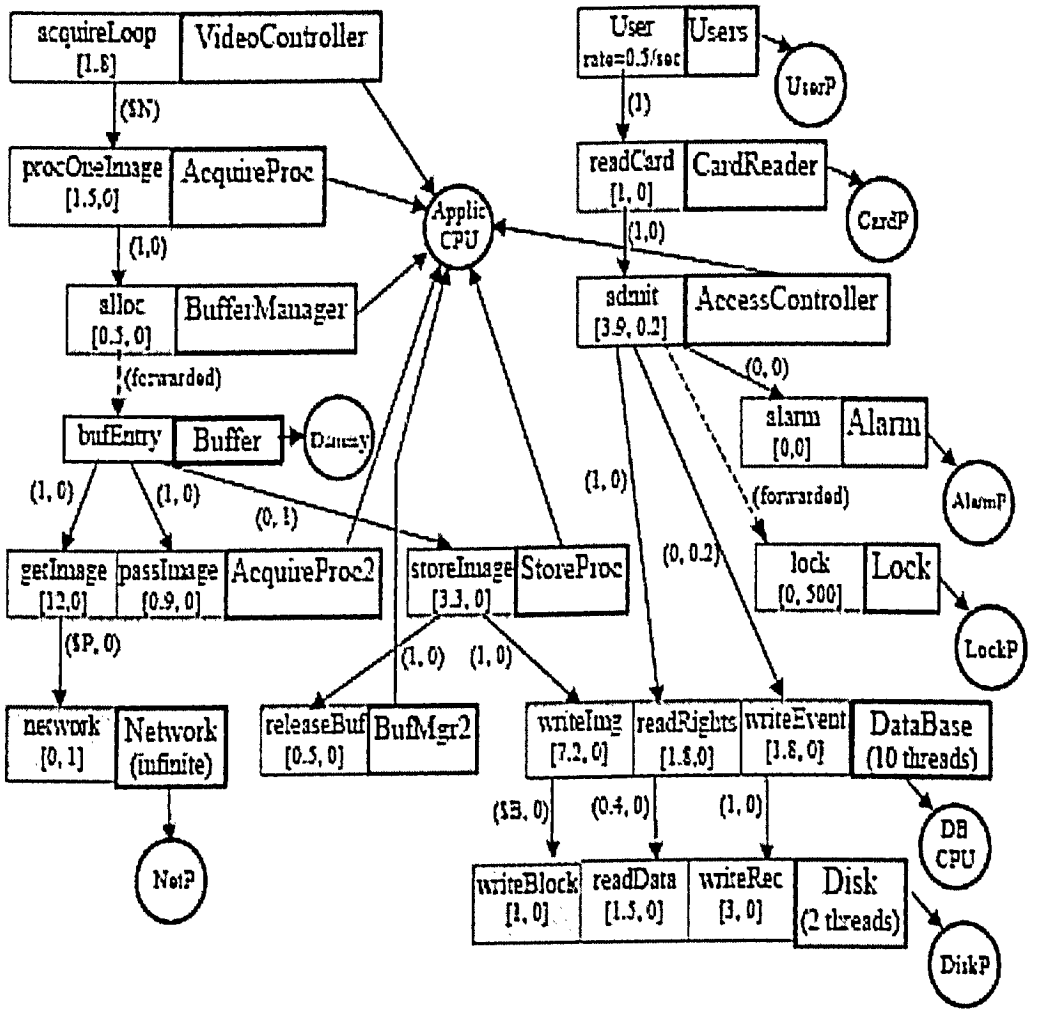

Figure 6.4 - LQN Model of Building Security System

In this system we add a secondary database server, called Database-2 (DB-2), as shown in figure 6.5. Thus if the primary database is unoperational, the AccessController task will use DB-2. The rest of the model and the mean number of calls are same as figure 6.4, except that storeImage will also use DB-2 when primary database is unoperational and DB-2 is allocated DB2CPU (processor).

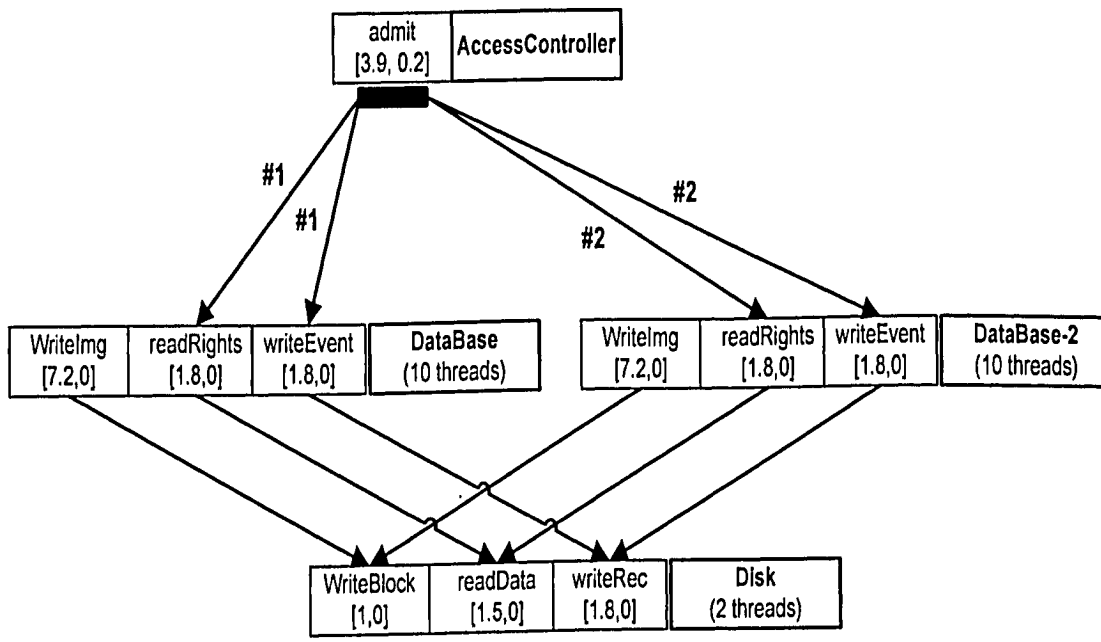

Figure 6.5 - FTLQN Model

6.4 Rejuvenated-FTLQN Model - Building Security System

The "Rejuvenated-FTLQN" model is applied for access control scenario which involves following software tasks: Users, CardReader, AccessController, Database and Disk. When the primary database is unoperational, the AccessController task will use the secondary database (DB-2). The failure profiles for the software tasks are as shown in the CTMCs below:

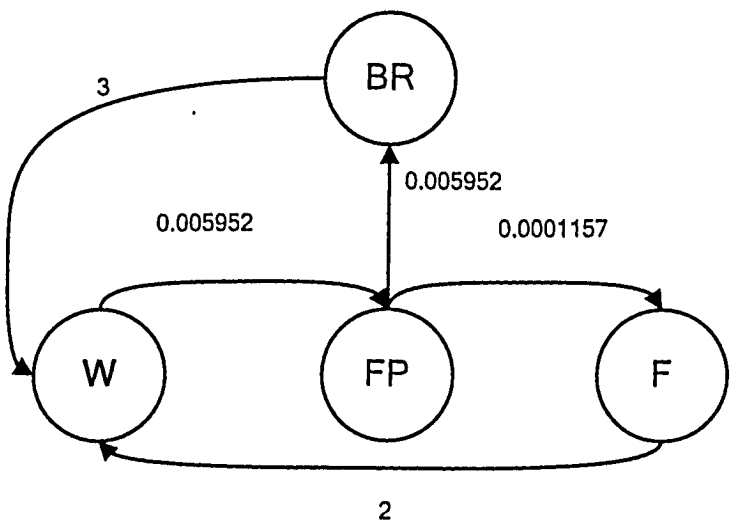

Figure 6.6 - CTMC for CardReader/Disk task

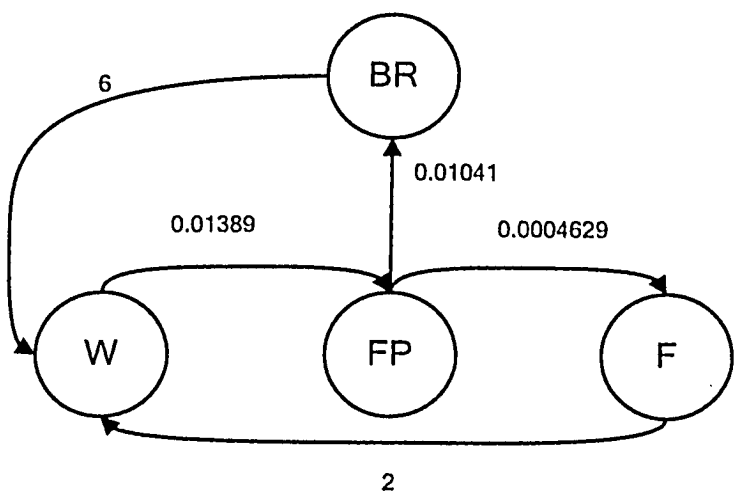

Figure 6.7 - CTMC for AccessController task

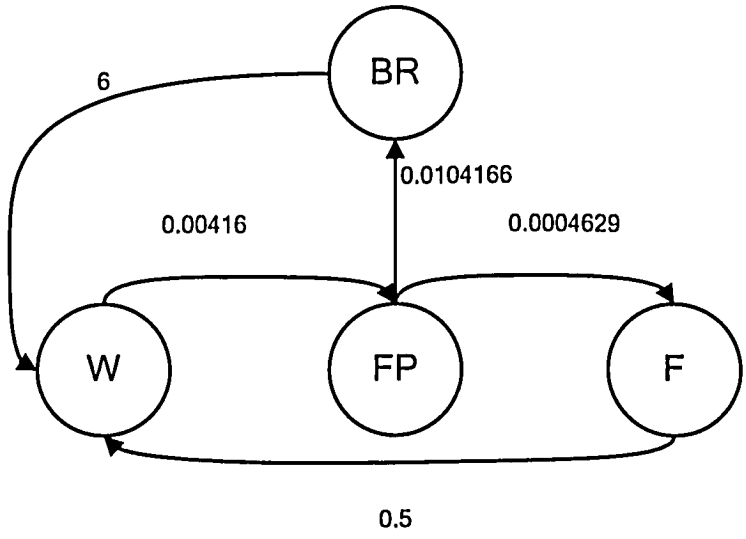

Figure 6.8 - CTMC for Database-1 and Database-2

The steady state probabilities obtained after solving the above CTMCs using the SHARPE tool is as shown in the table 14.

Table 14. Steady State Probabilities

Tasks	State F	State W	State FP	State BR
CardReader	0.000028	0.5043	0.4946	0.00098
AccessController	0.000129	0.4386	0.5603	0.00097
Database - 1	0.000255	0.7228	0.2764	0.00047
Database-2	0.000255	0.7228	0.2764	0.00047
Disk	0.000028	0.5043	0.4946	0.00098

The fault propagation AND-OR graph is shown in figure 6.9.

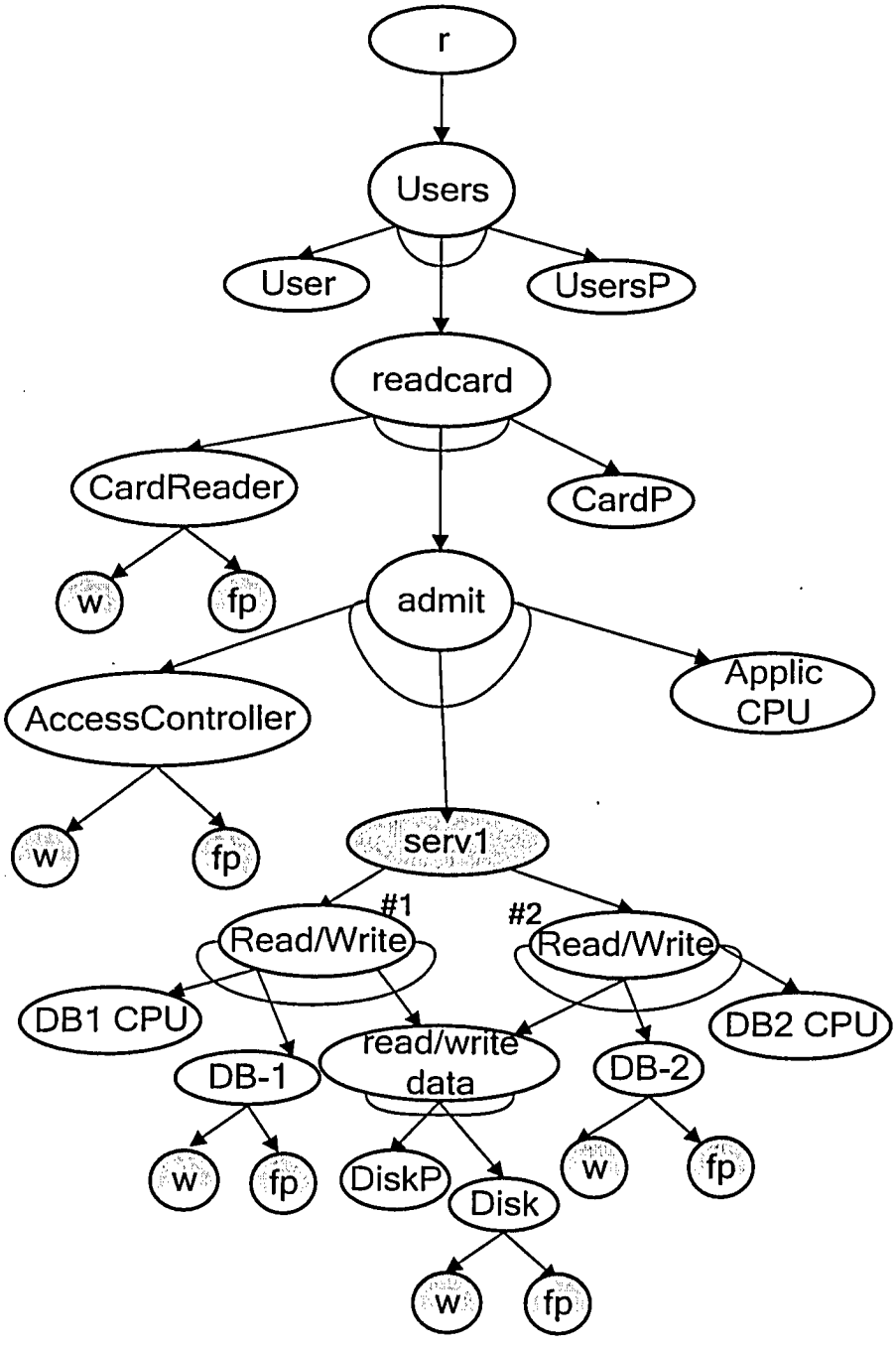

Figure 6.9 - Fault propagation And-Or graph

The next step is to generate the operational configurations of the system. The two operational configurations that arise due to presence of secondary server are shown in table 15.

Table 15. Operational Configurations (due to presence of secondary server only)

Notation	Operational Configurations
S1	CardReader-*W*, AccessController-*W*, Database-1- *W*, *Disk-W*
S2	CardReader-*W*, AccessController-*W*, Database-1- ***unoperational***, Database-2 – *W*, Disk-*W*

'*W*' indicates Working or highly robust state. S1 and S2 configurations arise due to presence of secondary server alone, without any consideration of software aging. For S1, we have 4 tasks and each task can be in highly robust state ('W') as well as failure probable state ('FP'). Thus we have 16 (2^4) different operational configurations including S1 for e.g. CardReader-*W*, AccessController-*W*, Database-1- *FP*, *Disk*-W is one of the possible operational configuration. Similarly for S2 also we have 16 different operational configurations including S2, for e.g. CardReader-*W*, AccessController-*FP*, Database-1- ***unoperational***, Database-2- *FP*, *Disk*-W is one of the possible operational configuration. **So finally we have 32 different operational configurations of the system.** The number of operational configurations is more in our case compared to when the performance degradation due to software aging is not considered (increases from 2 to 32). All the operational configurations are generated automatically by "Rejuvenated-FTLQNS" tool from the model description.

Thus we will have 32 different LQN Models corresponding to its equivalent operational configuration. The "Rejuvenated-FTLQNS" tool generates 32 different LQN input files and invokes LQNS [60] on each of the file. A PERL Script extracts the desired value (e.g. throughput) from the output file generated by LQNS. We are selecting throughput as

the reward rate to be associated with corresponding operational configuration. Throughput is high when the tasks are in highly robust state compared to when the tasks are in failure probable state due to aging for e.g. Operational configuration CardReader-*W*, AccessController-*W*, Database-1-*W*, *Disk*-*W* has a throughput of 0.126215/Sec, while operational configuration CardReader-*W*, AccessController-*W*, Database-1-*FP*, *Disk*-FP has 0.0969637/Sec. Similarly we will have different throughput value (reward rate) corresponding to every operational configuration.

Apart from the reward rate for every operational configuration, we also need to determine the probability of system being in each of that configuration in order to evaluate steady state performability. The failure probability of all the processors and reference task is assumed to be 0.05. To find the probability of operational configuration S1 following formula is used:

$$Prob\ (S1) = (1\text{-}F\_\{users\}) * (1\text{-}F\_\{UserP\}) * (P\_\{CardReader_w\}) * (1\text{-}F\_\{CardP\}) * (P\_\{AccessController_w\}) * (1\text{-}F\_\{ApplicCPU\}) * (P\_\{Database\text{-}1_w\}) * (1\text{-}F\_\{DBCPU\}) * (P\_\{Disk_w\}) * (1\text{-}F\_\{DiskP\}) = \mathbf{0.02239}$$

- $F\_\{users\}$ indicates failure probability for users task.
- $F\_\{UserP\}$ indicates failure probability for User processor which is used by users task.
- $P\_\{CardReader_w\}$ indicates probability of CardReader task is in state 'W'. This probability is obtained by solving the CTMC for CardReader task

All other parameters in the equation have the same interpretation.

Similarly for all other operational configurations related to S1, we can calculate the corresponding probability. If any particular task is in state 'FP' the steady state probability being in state 'FP' is obtained by solving CTMC for that task. Now to calculate the probability of operational configurations related to S2, we have to first determine the probability of primary database (DB-1) being unoperational. As all the 16 operational configurations related to S2 has primary database unoperational and thus

using secondary database. Figure 6.10 shows the MSFT which represents the combination of conditions that can cause Database-1 to be unoperational.

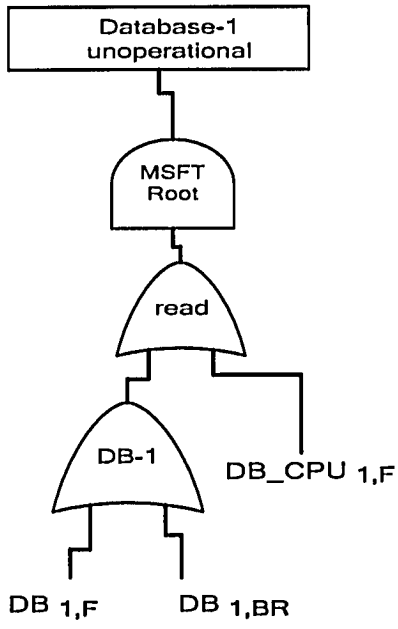

Figure 6.10 - MSFT for Database-1 (primary) unoperational

- DB $_{1,F}$ indicates probability of Database-1 being in 'F' (Failed) state.
- DB $_{1,BR}$ indicates probability of Database-1 being in 'BR' (Being Rejuvenated) state.
- DB_CPU $_{1,F}$ indicates probability of DB CPU (which DB-1 is using) being in 'F' (Failed) state.

To find the probability of operational configuration S2 following formula is used:

*Prob (S2) = (1-F_{users}) * (1-F_{UserP}) * (P_{CardReader$_w$}) * (1-F_{CardP}) * (P_ {AccessController$_w$}) * (1-F_{ApplicCPU}) * (P_{Database-2$_w$}) * (1-F_{DB2CPU}) * (P_{Disk$_w$}) * (1-- F_{DiskP}) * (P_{Database-1 unoperational}) =* **0.001146**

- *(P_{Database-1 unoperational})* is obtained by solving the MSFT shown in figure 6.10 using SHARPE.

- *Database-2* and its corresponding CPU (*DB2CPU*) are used in the formula.

Similarly we can calculate probability for all other operational configurations related to S2. Thus we will have probability of occurrence for all 32 different operational configurations. MSFT for System unoperational (for access control scenario) is shown in figure 6.11. The sum of the probabilities of all 32 operational configurations and system unoperational probability adds up to 1. MSFT for database-1 unoperational and system unoperational is automatically generated by "Rejuvenated-FTLQNS" tool from model description, and the required probability is extracted from the output generated by SHARPE.

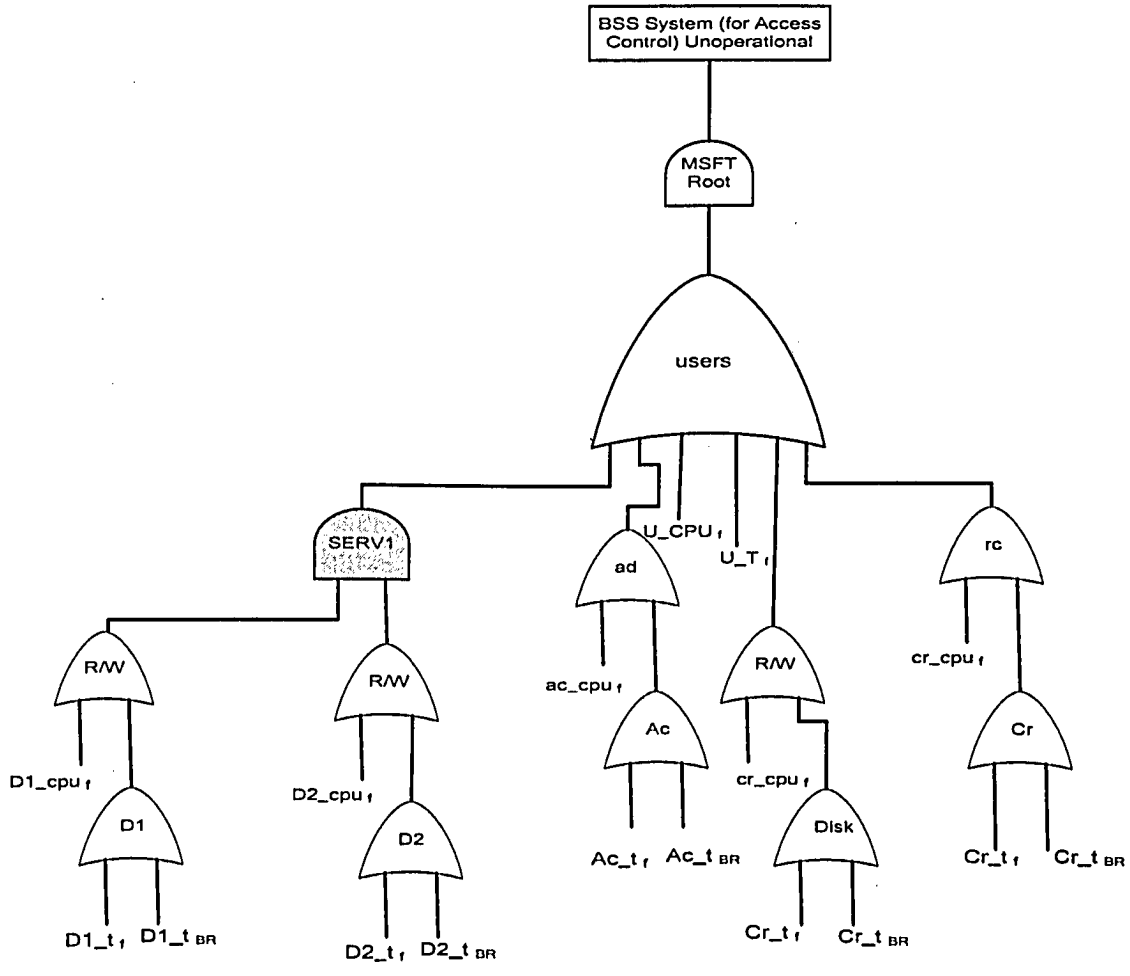

Figure 6.11 - MSFT for system unoperational

The steady state performabilty or expected steady state reward rate is calculated by multiplying the throughputs (rewards) associated with every operational configuration with their corresponding probabilities. We get the steady state performability of **0.06647** *requests/*sec.

6.5 Discussions

In this section we will discuss about (a) Model solution *with and without considering rejuvenation* for software tasks, comparing the steady state performability values when rejuvenation is performed compared to when it is not performed. (b) *Effects of changing the rejuvenation frequency (rate), repair rate after rejuvenation event and base longevity interval* on the steady state performability of the system.

6.5.1 Model Solution with and without Rejuvenation

This section describes the impact of performing rejuvenation on steady state performability. When rejuvenation is not implemented, the CTMC for the software tasks will be as shown in figure 3.2 of chapter 3. We will have same number of operational configurations as before (32), and the rewards (throughput) associated with the operational configurations will also remain the same. The difference is in the probabilities of occurrence for the operational configuration. When no rejuvenation is performed the software task will reside in failure probable state ('FP') for longer duration compared to when rejuvenation is performed. For example, say OC1 denote operational configuration- CardReader-*W*, AccessController-***FP***, Database-1- ***FP***, *Disk-**FP***, from table 16, we can see that when rejuvenation is implemented the system stays in this configuration for less amount of time compared to when rejuvenation is not implemented. OC2 denote CardReader-*W*, AccessController-***FP***, Database-1- *W*, *Disk-W*.

- **Without Rejuvenation:**
  Steady state performability= **0.0529726** *requests/*sec.

- **With Rejuvenation:**

  Steady state performability= **0.06647** *requests/*sec.

The result shows that the steady state performability increases when rejuvenation is performed. The increase is quite less because of the nature of the application but important point is that steady state performability increases. For other systems, with different set of parameters we may see a bigger difference.

Table 16. Probability with and without rejuvenation

Operational Configurations	Probability- **Rejuvenation Implemented**	Probability- **Rejuvenation not Implemented**
OC1	0.0446	0.5681
OC2	0.0366	0.0001428

6.5.2 Effects of Rejuvenation Rate

This section describes the effects of rejuvenation rate (frequency) on the steady state performability of the system. Rejuvenation rate refers to the rate R_4 in the CTMC shown in figure 6.12, and rate R_3 indicates the repair rate after rejuvenation event. We compare the effects of rejuvenation rate on two different designs:

(1) Access Control scenario with no fault tolerance.

(2) Access Control scenario with fault-tolerance.

In case (1) we don't have secondary database server available. So when the primary database server is unoperational (Undergoing Rejuvenation or Failed) the system in unoperational. But in case (2) when primary database server is unoperational, the system is still operational with AccessController task using secondary database. In case (1) we

have 16 different operational configurations, considering the fact that the software tasks have 2 operational states ('W' and 'FP'). In case (2) we have 32 different operational configurations as explained in section 6.4 of this chapter. We are increasing the rejuvenation rate for database servers and AccessController tasks. On the x-axis of the graph (figure 6.12) only rejuvenation rate for AccessController task is shown but the plot takes into consideration increase in rejuvenation rate for database servers too. All other model parameters are same as described in section 6.4 of this chapter. From figure 6.12, we can see that steady state performability is higher when we have secondary database available (with fault tolerance case). Initially when we increase the rejuvenation frequency, we see increase in steady state performability. But as soon as the rejuvenation frequency goes lower than 8 days (R_4= 0.041), there is a decline in steady state performability. Thus the increase in the value of steady state performability is only upto a point, after which it starts decreasing. "Rejuvenated-FTLQN" model can be used to study the effects of rejuvenation frequency on the steady state performability of the system and also to determine the threshold after which it has a negative impact.

If we increase rate R_3 and keep R_4 constant then the steady state performability increases as shown in figure 6.13. When we increase rate R_3, the software tasks stays in "Being Rejuvenated" state ('BR') for shorter duration and in highly robust state ('W') for longer duration, resulting in higher steady state performability. If we increase the rate at which the application goes from 'W' to 'FP' state (called the *base longevity interval*) and keep all other model parameters same, then the steady state performability decreases as shown in figure 6.14. Because when we increase the rate R_2, the software tasks stays in highly robust state for shorter duration and in failure probable state for longer duration (performing at degraded performance level for longer amount of time). Thus "Rejuvenated-FTLQN" model can also be used to analyze the effects of time taken to perform rejuvenation and base longevity interval on the steady state performability.

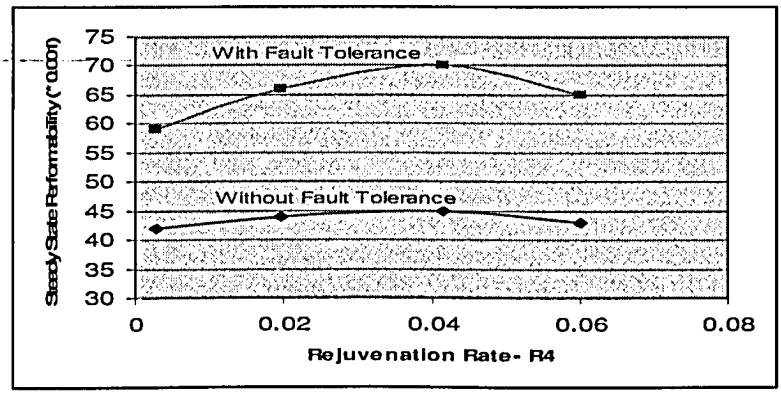

Figure 6.12 - Rate R_4 v/s Steady State Performability (SSP)

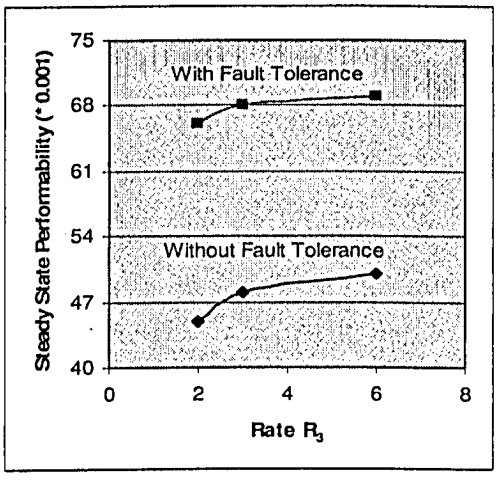

Figure 6.13 - Rate R_3 v/s SSP

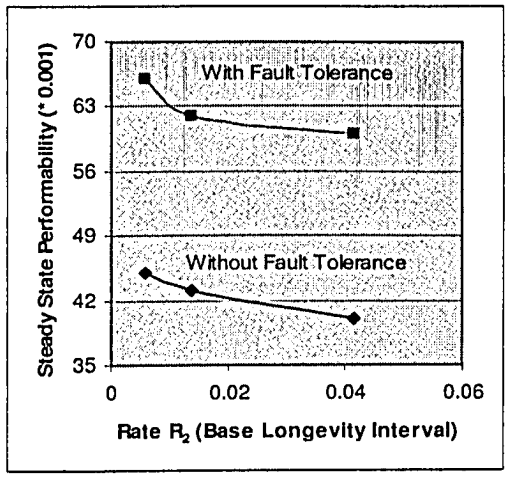

Figure 6.14 - Rate R_2 v/s SSP

6.6 Summary

In this chapter, the case study of large building security system was used to apply the "Rejuvenated-FTLQN" model for evaluating steady state performability. Two main scenarios were presented namely access control scenario and acquire/store video scenario and the model solution technique was shown for access control scenario. Finally the effects of rejuvenation frequency, time to perform rejuvenation and base longevity interval on steady state performability was discussed. This chapter shows that the Rejuvenated-FTLQN model can be used to study a substantially large system containing half a dozen components.

Chapter 7
Conclusions and Future Work

This research was motivated by the need for considering the effects of software aging and rejuvenation on the performability of the system. In this chapter the summary of the work, as well as the directions for future work is discussed.

7.1 Research Summary

This thesis has developed a model named as Rejuvenated-FTLQN to evaluate the steady state performability of fault tolerant layered distributed systems in the presence of software aging and rejuvenation on the servers at different layers. This approach avoids solving large markovian model. The existing FTLQN model was developed to express the layered service failure and repair dependencies. An algorithm was also provided for computing performability measures. However, in FTLQN model there was no consideration of software aging and rejuvenation phenomenon. But in many of the existing systems the server processes are intended to run continuously providing non-stop service and they may start aging after a certain period of time providing degraded service. Rejuvenation may be implemented to counteract the effects of aging. Rejuvenated-FTLQN model as described in chapter 3 was developed to take into account the aging and rejuvenation behaviour while evaluating performability.

Rejuvenated-FTLQN model solution uses LQN for performance analysis and MSFT for dependability analysis. A CTMC is added for every software task that describes different states in which the task resides and the transition between them. CTMC also includes the Failure Probable (FP) state due to aging and Being Rejuvenated (BR) state indicating that the task is undergoing rejuvenation and therefore unavailable. The CTMCs are solved using the SHARPE tool to obtain the steady state probabilities. These probabilities are further used in the dependability analysis part as an input to MSFT. The AND-OR graph representation of the Rejuvenated- FTLQN model is used to generate different

operational configurations of the system. Operational configurations of the system describe the structural variations caused due to fault tolerance and software aging. An LQN model (performance model) is constructed for each and every operational configuration and solved using the LQNS tool to obtain the desired performance measure. The aging of any particular task is modeled by changing the mean execution demand of its entries. To calculate the probability of occurrence of the operational configuration, a MSFT is used. A MSFT describes the combination of events that lead to the task or system being unoperational. For e.g. if any task is unoperational there are three possible causes: (1) the task is in failed state (2) the task is undergoing rejuvenation and (3) the processor that is used by the task is in failed state. The MSFTs are solved using the SHARPE tool to obtain the failure probability. The results from the performance analysis (LQN) as well as the dependability analysis (MSFT) are combined to obtain the steady state performability. It has been assumed here that the failures of the software tasks and processors are independent of each other

A software tool called Rejuvenated-FTLQNS (Fault Tolerant Layered Queueing Network Solver) was developed to automate the Rejuvenated-FTLQN model solution (chapter 5). Rejuvenated-FTQLNS has been developed in C++ and PERL. LQNS and SHARPE tools are invoked from Rejuvenated-FTLQNS tool. A model was solved for two cases (1) with rejuvenation implemented and (2) without rejuvenation and it was found that rejuvenation helps in improving the steady state performability of the system. The effects of rejuvenation frequency on the steady state performability was also studied and it was observed that increasing the rejuvenation rate results in higher steady state performability, but this was only upto a point. After which increasing the rejuvenation frequency had negative effects on steady state performability. A model was solved for a system without fault tolerance and a same system with fault tolerance (by the means of secondary server) and it was found that the steady state performability was higher for fault tolerant system. The effects of the time to perform rejuvenation and the time it takes to go from highly robust state to failure probable state due to aging (base longevity interval) on steady state performability was also studied (chapter 4). It was found that as

the time to perform rejuvenation as well as base longevity increases, the steady state performability decreases.

The Rejuvenated-FTLQN model was also used to analyze the large Building Security System (BSS). Access Control Scenario (to control access to a building) was described using the UML sequence diagram and used for the application of Rejuvenated-FTLQN model. The effects of performing rejuvenation and having a secondary database server were also discussed.

Rejuvenated-FTLQN model can be used to analyze the impact of software aging on any layer of the multi-tier application. As the processes running on the server are required to run non-stop for a long duration (in days, weeks or even years), they are prone to aging. The model proposed in this thesis can quantify the effects it has on the steady state performability of the system. In a way, a better picture is revealed. The pro-active approach to eliminate the effects of software aging can be implemented using software rejuvenation technique. The overhead involved with software rejuvenation is that the application may be unavailable during the time in which rejuvenation is performed. Rejuvenated-FTLQN model can be used to study the combined effects of aging and rejuvenation occurring at any layer, on the steady state performability of the system. Some of the other questions that can be answered using Rejuvenated-FTLQN model are: (1) How much gain in steady state performability can be obtained by implementing rejuvenation? (2) Increasing the rejuvenation frequency will increase the steady state performability by what amount? (3) How the actual time taken to perform rejuvenation and the base longevity interval are affecting the steady state performability of the system? (4) Adding a secondary server at which layer, will result in highest steady state performability value, considering the effects of software aging and rejuvenation?

7.2 Directions for Future Research

The work done in this thesis can be extended in following directions:

7.2.1 Reduction in the number of Operational Configurations

A drawback of the Rejuvenated-FTLQN model solution is the enormous number of operational configurations due to two operational states ('W' and 'FP') for each software task. This drawback could be addressed by approaching the problem from different perspectives. Some of them are as follows [16]:

1. One possibility might be to explore symmetry in redundancy. In this case, using the primary or the backup gives rise to the same performance model and thus the number of operational configurations can be reduced.

2. Another way is to approximate the solution by considering only those software tasks that affect the system in a greater amount and ignoring others. For example, the tasks which provide shared services are usually more sensitive than others since their failure triggers multiple failures. Identification of these components might be done by performing sensitivity analysis.

3. For any of the software tasks if we have the knowledge about degradation caused by software aging being small enough which results in both the states "W" and "FP" having approximately the same values. We can assume only for those particular tasks that degradation in performance is small enough to ignore. This assumption reduces the number of operational configurations by a great amount since many combinations of software task states can be aggregated to one operational configuration.

7.2.2 Time-Based Rejuvenation Policy

Rejuvenated-FTLQN model considers prediction-based policy in which the rejuvenation starts whenever a degraded state of the component is detected. Similar solution can also be developed for time-based rejuvenation policy. In time-based

rejuvenation policy the rejuvenation is performed after a specific time interval. So the only deciding factor in performing rejuvenation is time. For e.g. rejuvenation takes place after every 8 hours.

7.2.3 Semi-Markov Model for Modeling Aging and Rejuvenation

In Rejuvenated-FTLQN model, CTMC was used for capturing the effects of software aging and rejuvenation, in which the sojourn times in all states are exponentially distributed. For the non-exponentially distributed sojourn times, Semi-Markov model can be used for modeling aging and rejuvenation [19]. For Semi-Markov model the time spent in each state can follow any distribution.

7.2.4 Load-balanced Rejuvenated- FTLQN Model

In Rejuvenated-FTLQN model, it was assumed that the system is made fault tolerant with redundant server, meaning only when the primary server is unoperational, the secondary server will be used. The model solution can be extended for the system with load balanced between the servers. So both the servers will be used, with load balanced between them. The performability algorithm for load-balanced FTLQN model has been provided in [10]. This algorithm can be extended to include the effects of software aging and rejuvenation on servers.

Appendix A: BNF Grammar for Rejuvenated-FTLQNS Input File

This section describes the formal description of the Rejuvenated-FTLQNS input file grammar in BNF form. This grammar is an extension of the FTLQNS input file grammar provided in [21] and FTLQNS grammar is in turn an extension of LQNS grammar [10].

The extensions made to the FTLQNS grammar have been highlighted. The nonterminals are denoted by *<non-terminal_id>*, while the terminals are written without brackets.

The notation $\{\ldots\}_{n}^{m}$, where $n \leq m$ means that the part inside the curly brackets is repeated at least n times and at most m times. If $n = 0$, then the part may be missing in the input text.

General Information

⟨InputFile⟩ → *⟨generalInfo⟩ ⟨processorInfo⟩ ⟨taskInfo⟩ ⟨entryInfo⟩*

⟨generalInfo⟩ → *G ⟨comment⟩ ⟨convVal⟩ ⟨itLimit⟩ ⟨printInt⟩*$_{opt}$

⟨comment⟩ → *⟨string⟩ /* comment on the model */*

⟨convVal⟩ → *⟨real⟩ /* convergence value */*

⟨itLimit⟩ → *⟨integer⟩* /* max nb. of iterations */

⟨printInt⟩ → *⟨integer⟩* /* intermed. res. print interval */

⟨underrelaxCoef⟩ → *⟨real⟩* /* underrelaxation coefficient */

⟨endlist⟩ → -1 /* end of list mark */

⟨string⟩ → " ⟨text⟩ "

Processor Information

⟨*processorInfo*⟩ → P ⟨*np*⟩ ⟨*pDecList*⟩

⟨ *np*⟩ → ⟨*integer*⟩ /* total number of processors */

⟨*pDecl*List ⟩ → { ⟨*npDecl*⟩ }$_{0}^{np}$ ⟨*endList*⟩

⟨*pDecl* ⟩ → p ⟨*procId*⟩ ⟨*eschedulingFlag*⟩ ⟨*procQuantum*⟩$_{opt}$ ⟨*multiServerFlag*⟩$_{opt}$ ⟨*replicationFlag*⟩$_{opt}$ ⟨*procRate*⟩$_{opt}$ ⟨procFailprob⟩

⟨*procId*⟩ → ⟨*integer*⟩ | ⟨*identifier*⟩ /* processor identifier */

⟨*schedulingFlag*⟩ → f /* First come, first served */

| /* p priority, preemptive */

| /* n Head of line*/

| /* i Random */

| /* s Processor sharing */

⟨*procQuantum*⟩ → ⟨*real*⟩ /* processor time quantum */

⟨*multiServerFlag*⟩ → *m* ⟨*copies*⟩ /* number of duplicates* /

| /* i Infinite server */

⟨*replicationFlag*⟩ → *r* ⟨*copies*⟩ /* number of replicas* /

⟨*procRate*⟩ → *R* ⟨ratio⟩ /* Relative proc. speed* /

⟨procFailprob⟩ → (⟨*real*⟩ /* processor failure probability* /

⟨copies⟩ → ⟨integer⟩

⟨ratio⟩ → ⟨real⟩

Task Information

⟨taskinfo⟩ → T ⟨nt⟩ ⟨tDeclList⟩

⟨nt⟩ → ⟨integer⟩ /* total number of tasks */

⟨tDeclList⟩ → { ⟨tDecl⟩ }$_{0}^{nt}$ ⟨endList⟩

⟨tDecl⟩ → t ⟨taskId⟩ ⟨refTaskFlag⟩ ⟨entryList⟩ ⟨procId⟩
⟨taskPri⟩$_{opt}$ ⟨thinkTimeFlag⟩$_{opt}$ ⟨multiserverflag⟩$_{opt}$
⟨replicationflag⟩$_{opt}$ ⟨ctmcrate1⟩ ⟨ctmcrate2⟩ ⟨ctmcrate3⟩
⟨ctmcrate4⟩ ⟨ctmclambda⟩ ⟨agingpercent⟩$_{opt}$

⟨procId⟩ → ⟨integer⟩ | ⟨identifier⟩ /* task identifier */

⟨refTaskFlag⟩ → r /* reference task */
| /* n non-reference task */

⟨entryList⟩ → { ⟨entryId⟩ }$_{1}^{ne_t}$ ⟨endList⟩ /* task t has ne_t entries */

⟨entryId⟩ → ⟨integer⟩ | ⟨identifier⟩ /* entry identifier */

⟨taskPri⟩ → ⟨integer⟩ /* task priority, optional */

⟨thinktimeflag⟩ → z ⟨real⟩ /* task think time, optional */

⟨ctmcrate1⟩ → (⟨real⟩) /* ctmc rate from F (failed) to W (working) */

⟨ctmcrate2⟩ → (⟨real⟩) /* ctmc rate from W (working) to FP (aging) */

⟨ctmcrate3⟩ → (⟨real⟩) /* ctmc rate from BR (rejuvenated) to W (working) */

⟨ctmcrate4⟩ → (⟨real⟩) /* ctmc rate from FP (aging) to BR (rejuvenated) */

⟨*ctmclambda* ⟩ → (⟨*real*⟩) / * ctmc rate from FP (aging) to F (failed) */

⟨*agingpercent* ⟩ → ⟨ *integer*⟩ / * rate of performance degradation- to change the mean cpu demands of the entries, optional */

Entry Information

⟨*tDecl*⟩ → *E* ⟨*ne* ⟩ ⟨*entrydeclist* ⟩

⟨*ne* ⟩ → ⟨*integer*⟩ / * total number of entries */

⟨*entrydeclist*⟩ → { ⟨*entrydecl*⟩ }$_0^{ne\times 9}$ ⟨ *endList*⟩
/ * k = maximum number of phrases* /

⟨*taskinfo*⟩ → a ⟨*entryId*⟩ ⟨*arrivalRate*⟩

| c ⟨*entryId*⟩ { ⟨*coeffofVariation*⟩ }$_1^k$ ⟨*endlist*⟩

| f ⟨*entryId*⟩ { ⟨*phtypeFlag*⟩ }$_1^k$ ⟨*endlist*⟩

| s ⟨*entryId*⟩ { ⟨*serviceTime*⟩ }$_1^K$ ⟨*endlist*⟩

| y ⟨*fromentry*⟩ ⟨*toentry*⟩ { ⟨*rendezvous*⟩ }$_1^k$ ⟨*endlist*⟩

| ALT y ⟨*fromentry*⟩ [⟨*toPrioentryList*⟩]
{ ⟨*rendezvous*⟩ }$_1^k$ ⟨*endlist*⟩

⟨*ne* ⟩ → ⟨*real*⟩ / * open arrival rate to entry */

⟨*serviceTime* ⟩ → ⟨*real*⟩ / * mean phase service time */

⟨*phTypeFlag* ⟩ → ⟨*integer*⟩ / * 0 – stochastic phase, 1 – deterministic phase*/

⟨*coeffOfVariatio* ⟩→ ⟨*real*⟩ / * service time coeff. of variation */

⟨*rendezvous* ⟩ → ⟨*real*⟩ / * mean number of RNV s/ph */

⟨*fromEntry*⟩ → ⟨*entryId*⟩ /* source of a message */

⟨*toEntry*⟩ → ⟨*entryId*⟩ /* destination of a message */

⟨*toPrioEntryList*⟩ → $\{\langle entryId \rangle\}_1^{ne_{alt}}$ ⟨*endlist*⟩ /* ne_{alt} *is the number of alternative destinations of a message which are prioritized from left to right* */

Appendix B: Sample Input and output files generated by Rejuvenated-FTLQNS

This section describes the intermediate files that are generated by Rejuvenated-FTLQNS tool for the solution of model shown in figure 5.2. Figure B1 indicates the LQN input file generated by Rejuvenated-FTLQNS tool. Figure B1 corresponds to operational configuration: task1-*W*, task2-*FP*, task3-*FP*. LQNS tool in invoked to solve this LQN model and the output generated by LQNS is shown in figure B2. The PERL script shown in figure B3 extracts the throughput value from the output file generated by LQNS (Figure B2). This throughput value is then assigned as reward to that particular operational configuration.

```
G "generated for A1" 0.000001 50 5 0.900000 -1

P 3
p p1 f
p p2 f
p p3 f
-1

T 3
t t1 r e1 -1 p1
t t2 n e2 -1 p2
t t3 n e3 -1 p3
-1

E 3
s e1 0.000000 0.500000  -1
s e2 0.750000 0.000000  -1
s e3 0.750000 0.300000  -1
y e1 e2 1.000000 0.000000  -1
y e2 e3 1.000000 0.000000  -1
 -1
```

Figure B1. Sample LQN input file (in LQNS syntax)

```
0.lqn: warning: Entry "e1" has no service time specified for phase 1.
Copyright the Real-Time and Distributed Systems Group,
Department of Systems and Computer Engineering
Carleton University, Ottawa, Ontario, Canada. K1S 5B6

Generated by lqns, version 2.23 (Linux 2.6.20-1.2307.fc5smp.#1 SMP Sun
Mar 18 21:02:16 EDT 2007 i686)
Sat Jun  2 12:14:06 2007

Input:  0.lqn
Output: *stdout*
Command line: lqns
Comment: generated for A1


Convergence test value: 3.26986e-07
Number of iterations:   7

MVA solver information:
Layer    n    k srv     step()       mean      stddev     wait()
mean      stddev         User       System     Elapsed
1        7    1   2          37     5.2857    0.18443        591
84.429      6.0861  0:00:00.00  0:00:00.00  0:00:00.00
2       13    1   2          66     5.0769   0.076923       1008
77.538      2.5385  0:00:00.00  0:00:00.00  0:00:00.00
3        7    1   1          36     5.1429    0.14286        372
53.143      3.1429  0:00:00.00  0:00:00.00  0:00:00.00
Total   27    0   0         139     5.1481    0.06967       1971
73      3.1392  0:00:00.00  0:00:00.00  0:00:00.00

    User:      0:00:00.00
    System:    0:00:00.00
    Elapsed:   0:00:00.00


Processor identifiers and scheduling algorithms:

Processor Name  Type     Copies  Scheduling
p1              Uni      1       FCFS
p2              Uni      1       FCFS
p3              Uni      1       FCFS



Task information:

Task Name       Type     Copies  Processor Name  Pri Entry List
t1              ref        1     p1              0   e1(2 phases)
t2              serv       1     p2              0   e2
t3              serv       1     p3              0   e3(2 phases)
```

Entry execution demands:

Task Name	Entry Name	Phase 1	Phase 2
t1	e1	0	0.5
t2	e2	0.75	0
t3	e3	0.75	0.3

Mean number of rendezvous from entry to entry:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
t1	e1	e2	1	0
t2	e2	e3	1	0

Phase type flags:
All phases are stochastic.

Squared coefficient of variation of execution segments:
All executable segments are exponential.

Open arrival rates per entry:
All open arrival rates are 0.

Type 1 throughput bounds:

Task Name	Entry Name	Throughput
t1	e1	0.5
t2	e2	0.666667
t3	e3	0.952381

Mean delay for a rendezvous:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
t1	e1	e2	0	0
t2	e2	e3	0.0727273	0

Service times:

Task Name	Entry Name	Phase 1	Phase 2
t1	e1	1.57273	0.5
t2	e2	1.57273	0
t3	e3	0.75	0.3

Service time variance (per phase)
and squared coefficient of variation (over all phases):

Task Name	Entry Name	Phase 1	Phase 2	coeff of var **2
t1	e1	10.3493	0.25	2.46715
t2	e2	4.16567	0	1.68414

```
t3                  e3                  0.5625        0.09          0.591837


Throughputs and utilizations per phase:

Task Name           Entry Name          Throughput    Phase 1       Phase 2
Total
t1                  e1                  0.482456      0.758772      0.241228      1
t2                  e2                  0.482456      0.758772      0
0.758772
t3                  e3                  0.482456      0.361842      0.144737
0.506579


Utilization and waiting per phase for processor:  p1

Task Name           Pri n   Entry Name          Utilization Ph1 wait      Ph2
wait
t1                  0   1   e1                  0.241228    0             0


Utilization and waiting per phase for processor:  p2

Task Name           Pri n   Entry Name          Utilization Ph1 wait      Ph2
wait
t2                  0   1   e2                  0.361842    0             0


Utilization and waiting per phase for processor:  p3

Task Name           Pri n   Entry Name          Utilization Ph1 wait      Ph2
wait
t3                  0   1   e3                  0.506579    0             0
```

Figure B2. LQN Output file corresponding to Figure B1

```
#!/bin/perl

$takelqn=0;
$inputlqn=0;
open(RES, ">resultlqn2");
open (GETDATA, "<nooflqn2") or die("Error");
$number = <GETDATA>;

for ($count = $number; $count >=0; $count--)
{
system "lqns $inputlqn.lqn > $takelqn.out";
open(REWD, "$takelqn.out") || die "error in opening .out file!";
$takelqn=$takelqn+1;
$inputlqn=$inputlqn+1;
while(<REWD>)
      {
        $secmatch = index($_, "Throughputs ");
        if ($secmatch != -1)
        {
          $secmatch = index($_, "Throughput ");

          $thruput = substr( $_, $secmatch, 10 );
              print RES $thruput;
              print RES "\n";
              print "Value of thruput: $thruput";
          last;
        }
      }
      close( REWD );
}
close(RES);
close(GETDATA);
```

Figure B3. PERL Script to extract throughput

Figure B4 shows the input CTMC file corresponding to task 3 in FTLQN model shown in figure 5.2. SHARPE tool is invoked to solve this CTMC and the output generated by SHARPE is shown in figure B5. The output consists of steady state probabilities for each of the 4 states. The PERL script shown in figure B6 extracts the steady state probabilities value from the output file generated by SHARPE (Figure B2). This probability values are then used as an input to the MSFT as well to calculate the probability of operational configurations.

```
markov t3
0 1 2.000000
1 2 0.002976
2 0 0.000116
2 3 0.002976
3 1 3.000000
end

expr prob(t3, 0)
expr prob(t3, 1)
expr prob(t3, 2)
expr prob(t3, 3)
end
```

Figure B4. Sample CTMC input file (in SHARPE syntax)

```
prob(t3, 0):    2.8431e-05

-----------------------------------------

prob(t3, 1):    5.0930e-01

-----------------------------------------

prob(t3, 2):    4.9019e-01

-----------------------------------------

prob(t3, 3):    4.8627e-04
```

Figure B5. CTMC output file corresponding to Figure B4

```
#!/bin/perl

$take=1;
$input=1;
open(RES, ">resultct");
open (GETDATA, "<noofctmc") or die("Error");
$number = <GETDATA>;

for ($count = $number; $count >0; $count--)
{
system "/usr/local/bin/sharpe $input.ct > $take.ctout";
open(FP, "$take.ctout") || die "error in opening .ctout file!";
$take=$take+1;
$input=$input+1;
while (<FP>)
{
    $match = index($_, ":");
    if ($match != -1)
    {
      $grab = substr($_, 15, 11);
      print RES $grab;
        print RES "\n";
      print "Value of grab: $grab";
        print "\n";


    }
}
close(FP);
}
close(RES);
close(GETDATA);
```

Figure B6. PERL Script to extract steady state probabilities

Figure B7 shows the input MSFT file corresponding to task 3 being in unoperational state. SHARPE tool is invoked to solve this MSFT and the output generated by SHARPE is shown in figure B8. The PERL script shown in figure B9 extracts the failure probabilities value (probability at infinity) from the output file generated by SHARPE (Figure B2).

```
mstree entry
repeat task:0 prob(0.000028)
repeat task:3 prob(0.000486)

or t task:0 task:3
repeat processor:0 prob(0.050000)
or top t processor:0
end
cdf(entry,top)
end
-1
```

Figure B7. Sample MSFT input file (in SHARPE syntax)

```
CDF for system entry:

probability at 0:         5.0488e-02
probability at infinity: 9.4951e-01
continuous probability:   0.0000e+00
```

Figure B8. MSFT output file corresponding to Figure B4

```
#!/bin/perl

$takemsft=0;
$inputmsft=0;
open(RESMSFT, ">resultmsft");
open (GETDATAMSFT, "<noofmsft") or die("Error");
$number = <GETDATAMSFT>;

for ($count = $number; $count >0; $count--)
{
system "/usr/local/bin/sharpe $inputmsft.msft > $takemsft.msftout";
open(FPMSFT, "$takemsft.msftout") || die "error in opening .msftout
file!";
$takemsft=$takemsft+1;
$inputmsft=$inputmsft+1;
while (<FPMSFT>)
{
    $match = index($_, "probability at infinity:");
    if ($match != -1)
    {
      $grabprob = substr($_, 25, 11);
      print RESMSFT $grabprob;
        #print RESMSFT "\n";
      print "Value of grab: $grabprob";
        print "\n";


    }
}
close(FPMSFT);
}
close(RESMSFT);
close(GETDATAMSFT);
```

Figure B9. PERL Script to extract failure probability

Bibliography

[1] A. Artur, S.Luis, "*Deterministic Models of Software Aging and Optimal Rejuvenation Schedules*", 10th IFIP/IEEE International Symposium on Integrated Network Management, May 21 2007, pp.159-168.

[2] A. Avizienis, J. Laprie, and B. Randell, "*Dependability and its Threats: A Taxonomy*" in Building the Information Society: Proc, IFIP 18th World Computer Congress, 22-27 August 2004, Toulouse, France, R. Jacquart, Kluwer Academic Publishers, 2004, pp.91-120.

[3] A. Avritzer, E. Weyuker, "*Monitoring Smoothly Degrading Systems for Increased Dependability*", Empirical Software Eng. Journal, Vol 2, No 1, 1997, pp. 59-77.

[4] G. Bolch, S. Greiner, H. de Meer and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*, John-Wiley & Sons, 1998

[5] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 2nd edition, 2005.

[6] V. Castelli, R.E. Harper, P. Heidelberger, S.W. Hunter, K.Trivedi, K. Vaidyanathan, and W. Zeggert, "*Proactive Management of Software Aging*," IBM J. Research & Development, vol. 45, no. 2, Mar. 2001, pp. 311-332.

[7] K.Cassidy, K.Gross, A.Malekpour, "*Advanced Pattern Recognition for Detection of Complex Software Aging Phenomena in Online Transaction Processing Servers*", Proc. of the 2002 Int. Conf. on Dependable Systems and Networks, DSN-2002, pp. 478-483.

[8] G. Ciardo, R. Marie, B. Sericola and K. S. Trivedi, "*Performability Analysis Using Semi-Markov Reward Processes*", IEEE Transactions on Computers, vol. 39, no. 10, 1992, pp. 1251-1264.

[9] O. Das and C. M. Woodside, "*The Fault-tolerant layered queueing network model for performability of distributed systems*", IEEE Int. Computer Performance and Dependability Symposium (IPDS'98), 1998, pp. 132-141.

[10] O. Das, *Performance and dependability analysis of fault-tolerant layered distributed systems*, Master's thesis, Dept. of Systems and Computer Engineering., Carleton University, 1998.

[11] O. Das and C. M. Woodside, "*Evaluating layered distributed software systems with fault-tolerant features*", Performance Evaluation, 45 (1), 2001, pp. 57-76.

[12] O. Das and C. M. Woodside, "*Failure detection and recovery modelling for multi-layered service systems*", Fifth International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-5), Erlangen, Germany, Sept. 2001, pp. 131-135.

[13] O. Das and C. M. Woodside, "*Modeling the Coverage and Effectiveness of Fault-Management Architectures in Layered Distributed Systems*", IEEE International Conference on Dependable Systems and Networks (DSN'2002), June 2002, pp. 745-754

[14] O. Das and C. M. Woodside, "*Layered Dependability Modeling of an Air Traffic Control System*", IEEE Intl. Conference on Software Engineering (ICSE 2003) Workshop on Software Architectures for Dependable Systems, Portland, Oregon, USA, May 2003, pp. 50-55.

[15] O. Das and C. M. Woodside, "*Dependable LQNS: A Performability Modeling Tool for Layered Systems*", IEEE International Conference on Dependable Systems and Networks (DSN 2003), San Francisco, California, USA, June 2003, pp. 672.

[16] O. Das and C. M. Woodside, "*The Influence of Layered System Structure on Strategies for Software Rejuvenation*", Sixth Intl. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-6), Monticello, Illinois, USA, Sept. 2003, pp. 47-50.

[17] O. Das and C. M. Woodside, "*Computing the Performability of Layered Distributed Systems with a Management Architecture*", ACM Fourth International Workshop on Software and Performance (WOSP 2004), Redwood City, California, USA, Jan 2004, pp. 174-185.

[18] O. Das and C. M. Woodside, "*Analyzing the effectiveness of fault management architectures in layered distributed systems*", Performance Evaluation, 56, 2004, pp. 93-120.

[19] T. Dohi, K. Go˘seva–Popstojanova and K. S. Trivedi, "*Analysis of Software Cost Models with Rejuvenation*", In Proc.of the 5th IEEE Int.Symp.on High Assurance Systems Engineering, HASE 2000, Albuquerque, NM, November 2000, pp. 25-34.

[20] T. Dohi, K. Go˘seva–Popstojanova and K. S. Trivedi, "*Statistical Non-Parametric Algorithms to Estimate the Optimal Software Rejuvenation Schedule*", Proc.of the 2000 Pacific Rim Int.Symp.on Dependable Computing, PRDC 2000, Los Angeles, CA, December 2000, pp. 77-84.

[21] G. Franks, *Performance Analysis of Distributed Server Systems*, Ph.D. thesis, Dept. of Systems and Computer Engineering., Carleton University, 1999

[22] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and C. M.Woodside, "*Performance Analysis of Distributed Server Systems*," in 6th Intl. Conf. on Software Quality (6ICSQ), Ottawa, 1996, pp. 15-26.

[23] S. Garg, A. van Moorsel, K. Vaidyanathan, K. Trivedi, "*A Methodology for Detection and Estimation of Software Aging*," Proc. Ninth Int'l Symp. Software Reliability Eng., Nov. 1998, pp. 282-292.

[24] S. Garg, A. Puliafito, M. Telek and K. S. Trivedi, "*Analysis of Software Rejuvenation Using Markov Regenerative Stochastic Petri Net*", In Proc.of the Sixth Int.Symp. on Software Reliability Engineering, Toulouse, France, October 1995, pp 180-187.

[25] S. Garg, Y. Huang, C. Kintala and K. S. Trivedi, "*Time and Load Based Software Rejuvenation: Policy, Evaluation and Optimality*", Proc. of First Fault Tolerant Symposium, India, December 1995, pp. 22-25

[26] S. Garg and A. P. A. Van Moorsel, "*Towards performability modeling of software rejuvenation*", presented in PMCCS, Bloomingdale, Illinois, September 6-9, 1996.

[27] S. Garg, A. Puliafito, M. Telek and K.S. Trivedi, "*Analysis of. Preventive Maintenance in Transactions Based Software Systems*", IEEE Trans on computers, vol 47, no 1, 1998, pp 96-107

[28] J. Gray, "*Why Do Computers Stop and What Can Be Done About It?*" Proc. Fifth Symp. Reliability in Distributed Software and Database Systems, Jan. 1986, pp. 3-12

[29] M. Grottke & Kishor S Trivedi, "*Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate*", IEEE Computer Magazine (Feb 2007), pp. 107-109.

[30] B. Haverkort, R. Marie, G. Rubino and K. S. Trivedi, *Performability Modelling: Techniques and Tools*, John Wiley and Sons, Chichester, England, April 2001.

[31] Y. Huang, C. Kintala, N. Kolettis and N. D. Fulton, "*Software Rejuvenation: Analysis, Module and Applications*", in Proc. 25th Annual Intl. Symp. on Fault Tolerant Computing, 1995, pp. 381-390.

[32] *IBM Netfinity Director Software Rejuvenation* - White Paper. IBM Corp., Research Triangle Park, NC, Jan 2001.

[33] L. Jiang, G. Xu, "*Modeling and analysis of software aging and software failure. Journal of Systems and Software*", 2007, pp. 590-595.

[34] P. Jogalekar, Murray Woodside, "*Evaluating the Scalability of Distributed Systems*", IEEE Trans. on Parallel and Distributed Systems, v 11 n 6, June 2000, pp. 589-603.

[35] E. Lazowska, John Zahorjan, G. Scott Graham, Kenneth C. Sevcik, *Quantitative System Performance*, Prentice-Hall, Inc., 1984.

[36] L. Li, K. Vaidyanathan and K. S. Trivedi, "*An Approach to Estimation of Software Aging in a Web Server*", In Proc.of the Int.Symp.on Empirical Software Engineering, ISESE 2002, Nara, Japan, October 2002, pp. 91-103.

[37] E. Marshall, "*Fatal Error: How Patriot Overlooked a Scud*", Science, Volume 255, Issue 5050, pp. 1347.

[38] J. F. Meyer, "*On Evaluating the Performability of Degradable Computing Systems*", IEEE Trans. on Computers, vol. 29, no. 8, Aug 1980, pp. 720-731.

[39] J. F. Meyer, "*Performability: A retrospective and some pointers to the future*", Journal Performance Evaluation. 14(1992), pp. 139-156.

[40] R. Mendel and Ousterhout, K. John, "*The Design and Implementation of a Log-Structured File System*". ACM Transactions on Computer Systems, Vol. 10 Issue 1, pp. 26-52.

[41] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, California, 1980.

[42] T. Omari, Greg Franks, Murray Woodside, Amy Pan, "*Solving Layered Queueing Networks of Large Client Server Systems with Symmetric Replication*", Proc. 5th Int. Workshop on Software and Performance (WOSP 2005), July 2005, pp 159-166.

[43] D. L. Parnas, "*Software aging*", In Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994, pp. 279-287.

[44] A. Pfening, S. Garg, A. Puliafito, M. Telek, K.Trivedi , "*Optimal Software Rejuvenation for Tolerating Soft Failures*", Performance Evaluation, Vol 27 & 28, Oct 1996, North-Holland, pp 491-506.

[45] J. A. Rolia and K. C. Sevcik, "*The Method of Layers*", IEEE Trans. on Software Engineering, vol. 21, no. 8, August 1995, pp. 689-700.

[46] K. Rinsaka and T. Dohi, "*Behavioral Analysis of a Fault-Tolerant Software System with Rejuvenation*", IEICE Trans D: Information, December 1, 2005; E88-D (12), pp. 2681-2690.

[47] R. A. Sahner, K. S. Trivedi and A. Puliafito (1996), *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*, Kluwer Academic Publishers, Boston.

[48] L. Silva, H. Madeira, G. Silva, "*Software Aging and Rejuvenation in a SOAP-based Server*," nca, Fifth IEEE International Symposium on Network Computing and Applications (NCA'06), 2006, pp. 56-65

[49] R. M. Smith, K. S. Trivedi and A. V. Ramesh, "*Performability Analysis, Measures, an Algorithm and a case study*", IEEE Trans. on Computers, vol. 37, no. 4, 1988, pp. 406-417.

[50] C.U. Smith, *Performance Engineering of Software Systems*, Addison Wesley, 1990.

[51] F.Sheikh and C.M. Woodside, "*Layered Analytic Performance Modelling of a Distributed Database System*", *Proc. 1997 International Conf. on Distributed Computing Systems*, May 1997, pp. 482-490.

[52] A.Tai, S.Chau, L.Alkalaj, H.Hecht. "*On-board Preventive Maintenance: Analysis of Effectiveness an Optimal Duty Period*", Proc. Third International Workshop on Object-Oriented Real-Time Dependable Systems, Feb. 1997, pp. 40-47.

[53] K Trivedi, "*SHARPE 2002: Symbolic Hierarchical Automated Reliability and Performance Evaluator*", IEEE Int. conf. On Dependable Systems and Networks, 2002, pp. 23-26.

[54] K. S. Trivedi, *Probability and Statistics, with Reliability, Queuing and Computer Science Applications*, 2nd edition. John Wiley, 2001.

[55] K. S. Trivedi, G. Ciardo, M. Malhotra and R. A. Sahner, "*Dependability and Performability Analysis*", in Performance Evaluation of Computer and Communication Systems, LNCS, L. Donatiella, R. Nelson (eds.), Springer-Verlag, 1993, pp. 587-612.

[56] K.S.Trivedi, K.Vaidyanathan, and K.Goseva- Postojanova, "*Modeling and Analysis of Software Aging and Rejuvenation*", Proc. 33rd Annual Simulation Symp., IEEE Computer Society Press (2000), pp. 270-279.

[57] K. Vaidyanathan, K.Trivedi,, "*A comprehensive model for software rejuvenation*" IEEE Trans. on Dependable and Secure Computing, 2(2), 2005, pp. 124 – 137.

[58] K. Vaidyanathan, R. E. Harper, S. W. Hunter and K.Trivedi, "*Analysis and implementation of software rejuvenation in cluster systems*", ACM SIGMETRICS/Performance 2001, pp. 62-71.

[59] D. Wang, W. Xie, K. S. Trivedi, "*Performability analysis of clustered systems with rejuvenation under varying workload* ", Perform. Eval. 64(3), 2007, pp. 247-265.

[60] C. M. Woodside, J. E. Neilson, D. C. Petriu and S. Majumdar, "*The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software*", IEEE Trans. on Comp, 44(1), 1995, pp. 20-34.

[61] C. M. Woodside, "*Performability modelling for multi-layered service systems*", Third International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-3), Illinois, USA, Sept. 1996.

[62] C. M. Woodside, "*Layered Resources, Layered Queues and Software Bottlencks*", a tutorial presented to the 2003 Illinois Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems, Sept 2, 2003.

[63] J. Xu, C.M.Woodside, D.Petriu "*Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time*", Proc. 13th Int Conf. on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS 2003), Urbana, Illinois, USA, Sept 2003, vol. LNCS 2794, Lecture Notes in Computer Science, Springer-Verlag, pp. 291-310.

[64] X. Zang, D. Wang, H.,Sun, K. Trivedi, "*A BDD-based algorithm for analysis of multistate systems with multistate components*", IEEE Trans. on Computers, 52(12), 2003, pp.1608-1618.

[65] http://www.apache.org [Online]

[66] http://httpd.apache.org/docs/1.3/misc/perf-tuning.html [Online]

[67] http://srejuv.ee.duke.edu/ [Online]

[68] http://www.doc.ic.ac.uk/~nd/surprise_95/journal/vol4/eaj2/report.html [Online]