THE EFFECTS OF A FAULT MANAGEMENT ARCHITECTURE ON THE PERFORMANCE OF A CLOUD BASED APPLICATION

by

Ghazal Zamani

B.A.Sc., Concordia University, 2014

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, 2017

Author’s Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

THE EFFECTS OF A FAULT MANAGEMENT ARCHITECTURE ON THE PERFORMANCE OF A CLOUD BASED APPLICATION

Ghazal Zamani

Master of Applied Science (M.A.Sc.)

Electrical and Computer Engineering

Ryerson University, 2017

Abstract

Increasingly, the application providers are using a separate fault management system that offers out-of-the-box monitoring and alarms support for application instances. A fault management system usually consists of a set of management components that does both fault detection and can trigger actions, for example, automatic restart of monitored components. Such a distributed structure supports scalability and helps to ensure that an application meets its quality requirements. However, successful recovery of an application now depends on the fault management architecture and the status of the management components. This thesis presents a model that accounts for the effect of management-architecture based coverage on the mean throughput of an application. Such a model would benefit the application providers for choosing the right fault management architecture for their applications. Comparing five different sample fault management architectures, shows that for higher workload, the case with highest number of detection paths has the maximum throughput.

Acknowledgements

I would like to thank Prof. Olivia Das for her endless academic and financial support during my research and study, and provisions above the academic contexts. Her limitless support and effective mentorship has been essential to the successful completion of this thesis. I would also like to thank my thesis defense committee Prof. Vadim Geurkov, Prof. Cungang (Truman) Yang, and Prof. Karthi Umapathy for their valuable time and effective comments. Finally, I reserve special thanks for my family, for their boundless mental, emotional, and financial support during my hard work.

Dedication

To my beloved parents for their passionate encouragements and generous financial support that made it possible for me to achieve academic success.

To my brother for being a great source of motivation and inspiration.

To my husband, the love of my life, for his meaningful assistance, tireless guidance and endless patience, without whom none of this would have been possible.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ArvRt	Arrival Rate
DES	discrete event simulation
FDR	Failure-Detected-and-Restarted
FIFO	First-in, First-out
FND	Failure-Not-Detected
IaaS	Infrastructure-as-a-Service
LB	Load Balancer
NTL	Normalized Throughput Loss
PaaS	Platform-as-a-Service
QoS	Quality of Service
Resptm	Response Time
RT	Response Time
SaaS	Software-as-a-Service
SEM	Standard Error of Mean
SLA	Service Level Agreement
SoA	Service Oriented Architecture
SPN	Stochastic Petri Net
SRN	Stochastic Reward Net
Stdev	Standard Deviation
THR	Throughput
VM	Virtual Machine

1. Introduction

1.1. Motivation

Application providers are increasingly leveraging cloud computing resources to enhance the scalability of services, provided by their applications, delivered to their end users. Cloud computing offers several benefits to its consumers, including dynamic scaling of VMs as required on a pay-per-use basis, and consistent performance and availability of the virtual computing resources [1].

In order to improve performance, application providers often employ load-balancing replication [2]–[5] for their applications. Such replication distributes the workload across multiple application instances where each instance usually runs on its own virtual machine (VM). An application can still function in response to application instance failures, perhaps with a degraded performance. However, the replication is ineffective if the mechanisms are not in place to detect and recover from failures.

Nowadays, the application providers are using a separate fault management system [6] that offers out-of-the-box monitoring and alarms support for the application components (instead of implementing the failure detection mechanisms within the application itself). Such usage promotes software reuse and additionally eases the development of the application. A fault management system is usually distributed in

nature and consists of a set of management components that does both fault detection and can trigger actions, for example, automatic restart of the monitored components. This sort of distributed structure supports scalability and helps to ensure that an application meets its quality requirements. However, successful recovery of an application now depends on the fault management architecture and the status of the management components. Here, recovery implies isolating and removing the failed application instance(s) and re-distributing the load among the operational instances.

In order to evaluate the mean throughput of these applications, one must consider the fraction of the jobs that are lost, also known as *normalized throughput loss* [7]. When an application instance fails, jobs are lost because the load balancer does not know about the failure and it continues to send the jobs to the failed instance. This may happen due to following reasons:

1. The management component(s) responsible for notifying the load balancer about the failure are in failed status, or;
2. The management component(s) responsible for detecting the failure are in failed status, or;
3. The failure is currently being detected and the load balancer is yet to know about the failure.

The performance measures of systems in presence of failures, also known as *performability measures*, can be evaluated analytically using an exact monolithic Markov model [8]. This approach suffers from largeness and stiffness problem. It can also be

evaluated using hierarchical Markov Reward models [8]: a higher-level Markov dependability model and a set of lower level performance related models, one for each state in the dependability model. The use of Markov Reward models is limited to scenarios where the failure and restarts occur at much slower time scale than the processing times. This limits its use for the current work since the restart times and failure detection times may be similar in time scales as the application processing times.

This thesis presents a discrete-event simulation model that accounts for the management-architecture-based coverage on the mean throughput of an application. It considers the job loss occurring due to the above-mentioned reasons. Such a model would benefit application providers in choosing the right fault management architecture for their applications. Although the proposed simulation model may be computationally expensive as compared to an analytical counterpart, it is more general in terms of service time, inter-arrival time, failure time, detection time, and restart time distributions.

1.2. Cloud Computing

In recent years, cloud computing has emerged as a popular computing service environment for personal, academic, and industrial applications, including Google, Amazon, and Baidu. Cloud computing is an internet based computing technique, which uses a network of remote servers to manage, store, and process the shared information and resources. This technology provides enhanced computing efficiency by centralizing data storage and data processing. The most significant advantages of cloud computing include cost efficiency by reducing the software and hardware costs, time efficiency by

reducing the computation times, improved accessibility providing access to data and resources at any time and any location, and improved data recovery. The primary goal of cloud computing is to make efficient access available to remote and geographically distributed resources. In order to distribute the cloud resources efficiently and ensure providing the users with necessary resources, it is required to implement optimized resource management and provisioning strategies in cloud systems. Cloud resource provisioning and management is accomplished by using different strategies, including: dynamic provisioning and user-self provisioning. The fundamental of all these strategies is to achieve fair distribution of resources while maintaining maximum utilization of the resources and maintaining minimum resource idle time.

A key component in the cloud service provisioning is the Service Oriented Architecture (SOA) [9], which captures computational resources through abstract interfaces to separate services from their corresponding implementations. Conventional cloud service models include Infrastructure-as-a-Service (IaaS), Platform-as-a-service (PaaS), and Software-as-a-Service. IaaS, formerly known as Hardware-as-a-Service, allows the customers to access the resources, such as storage, processing, and networking, remotely over the internet. This enables the client to dynamically scale the configuration and only pay for those services that are actually employed. IaaS is offered in three different models, namely private, public, and hybrid. PaaS deals with operating systems and delivers a platform that allows the users to execute existing applications or to develop and test new application by providing the customer with virtualized servers. This is especially beneficial for geographically-distributed development teams to work on a

software development project together. SaaS is the most popular model of using cloud computing, which allows several clients to share infrastructure in a private and secure manner.

1.3. Software as a Service (SaaS)

An application provides certain types of service to its end users. An application is an example of a SaaS software owned by an application provider who chose to deploy it in a cloud. SaaS is a method of software delivery and licensing in which a third-party provider hosts applications and makes them available to clients remotely over the Internet. SaaS is the application layer of cloud computing along with the platform layer, platform as a service (PaaS), and the infrastructure layer, infrastructure as a service (IaaS).

Unlike the conventional on-premise software delivery model, in SaaS software distribution model a third party hosts and maintains the software application and data. This eliminates the need for companies to install and run software applications locally. This in turn significantly shrinks the extensive hardware acquisition expenses as well as provisioning and maintenance, software licensing, installation, and support costs. Recently the application of SaaS delivery model is increasing in healthcare and engineering practices. Examples of SaaS include Google Doc, Gmail, and Zoho.

The main benefits of the SaaS delivery model include:

a. SaaS facilitates the pay-as-you-go model, where instead of purchasing the software license and the corresponding supporting software, it offers the customers a

subscription (either monthly or annually), which allows them to access the applications and data, as well as providing the associated support and maintenance.

b. SaaS provide remote access to the software application and data over the internet, which allows the customers to access them from any physical location.

1.4. Research Objective

This thesis develops a model that accounts for the effect of management-architecture based coverage on the mean throughput of an application. The value of including the fault management architecture in the analysis is first to account for the failures and restarts of managers, second is to include delays to detect the failures, and third is to evaluate the limitations of the fault management architecture. These three considerations increase the fraction of jobs that are lost thereby affecting the system throughput. This thesis demonstrates the use of the model by comparing five different sample fault management architectures. The application model and its fault management architecture models are simulated by using a discrete event simulation framework called SimPy, a Python based framework. This fault management model analyzes cases with one or more points of failure. There are number of application servers running on their virtual machines.

The examples given here consider only failures of application instances and managers. This is because, usually, the virtual machines and physical machines are highly available and they are usually managed by the cloud providers. However, if the application provider also wants to monitor the virtual machines and/or physical machines

allocated for their applications, then that can be easily included in the analysis as well. Different types of failure can be considered including, crash and sudden failures, which occur randomly, in addition to, security or aging failures that happen when the component gets obsolete and crashes. This work only considers the crash and sudden failures.

The proposed model will benefit application providers to answer several important questions related to selection of fault management architecture for their application. For example, if I buy three managers (with given failure and restart rates) to monitor my application instances, how should I connect them so that it meets the throughput and response time SLAs? What detection interval should I set while configuring the managers?

In this work, five different sample fault management architectures are investigated. For ***Case-1***, it is assumed that the application server's VMs can fail, however, no management component failure occurs at any point of time. For this case, the architecture is constructed such that each management component monitors one application server, and a client manager controls all the management components. The general structure of ***Case-2*** is very similar to Case 1, except that management components can also fail in this case. Since there is no manager monitoring the managers, in Case 2, the failure of managers must be detected by human administrators. ***Case-3*** represents a multi-layered management architecture. In this case, some of the mangers may monitor more than one application server, such that other than the client manager

that acts as a central manager, there are other management components additionally monitoring their status. In ***Case-4*** each manger not only monitors its assigned application servers, but also monitors the other management components, and reports back to the client manager. Finally, for ***Case-5***, every application server is monitored by all the management components. Additionally, similar to Case 4, each manager monitors all the other management components.

Subsequently, average response time, job loss probability, arrival rate, and throughput are determined for each case for multiple runs of the SimPy simulation model. Different parameters are considered in the simulations, including change in detection time, failure time, and inter arrival time. Finally, the results of the simulations for the proposed five architecture cases are compared and discussed.

1.5. Research Contributions

The contributions of this thesis are as follows:

a. Development of a model that accounts for the effect of fault management-architecture based coverage on the mean throughput of an application.
b. Discrete event simulation of the proposed model using SimPy framework.
c. Comparison of five different fault management architectures on the performance of a cloud application using the proposed modeling technique.

The results of this analysis are incorporated into a research manuscript and submitted to the 13th European Dependable Computing Conference (EDCC 2017) in Geneva, Switzerland [10].

1.6. Thesis Outline

This thesis is organized as follows*: Chapter 2* represents the background on performability models. The analytical and simulation approaches are described and the benefits of the simulation approaches over the analytical counterparts are discussed to justify the application of simulation approaches in this work. *Chapter 3* describes the proposed modeling technique. *Chapter 4* compares the impact of five different fault management architectures on the performance of a cloud application. Finally, *Chapter 5* provides the summary of the results of the thesis.

2. Background

This chapter discusses the key principles and definitions that are applied to this work. Different performability evaluation techniques are described and the advantages and the shortcomings of each technique is discussed.

2.1. Performability Evaluation Techniques

There are two ways to evaluate performability of a system: measurement-based evaluation or model-based evaluation.

2.1.1. Measurement-based Evaluation

Several cloud service evaluations are based on measurements performed on a cloud infrastructure as a testbed. The general procedure includes specifying the purpose and the scope of the evaluation and identifying the features/aspects of the cloud to be evaluated, classifying the performance metrics and indicating the proper benchmarks applications for testing, and finally, setting up the experimental environment. A list of general performance metrics for evaluating typical cloud services is given in [11]. Stantchev [12] proposed a general approach based on architectural transparent black-box methodology for evaluating non-functional QoS properties of individual cloud services. Atas and Gungor [13] proposed a framework for PaaS performance assessment based on a set of benchmark algorithms that allows the computation of the most proper PaaS service provider based on different source and application requirements. The main

disadvantage of measurement-based cloud evaluation approaches is that it is typically expensive to develop large scale testbeds that realistically demonstrate cloud service provisioning scenarios. Furthermore, they typically necessitate costly and widespread measurements and experimentations and the accuracy results significantly rely on the design.

2.1.2. Model-based Evaluation

The objective of performability modeling and assessment is to provide insight into systems that either are not built yet, or are performing under certain conditions where they are not accessible for measurements or fail quite intermittently. The most common approaches for solving models are analytical and simulation techniques. In analytical approaches, certain boundaries are applied to the models to ensure the existence and possibility of analytical solutions. In general, the analytical approaches are classified into two categories: closed form, where an explicit expression is derived to describe the measure of interest in terms of model parameters and structure, and numerical, where a system of equations is solved by applying numerical techniques, including iterative procedures, to determine the measure of interest. Alternatively, simulation approaches emulate the behavior of the system by executing an appropriate simulation program, which provides statistical estimates of the measurement of interest. In general, analytical techniques are superior in terms of computational efficiency, however, they are applicable to a restricted set of models, making simulation approaches generally more applicable.

2.2. Performability Modeling Techniques

Pure performance analysis of systems generally over-estimates the ability of the system to perform a particular job. In contrast, pure reliability/availability analysis is typically too conservative, since the performance considerations are disregarded. Consequently, in order to model a system appropriately, it is required to develop modeling techniques that combine the system performance and reliability, which is known as performability [14].

2.2.1. Analytical-Model Based Techniques

Different types of analytical models exist to determine the performability of a system, including Monolithic models and Hierarchical models. Figure 2.1 demonstrates the typical analytical performability approaches. The monolithic approach [15] combines the performance and reliability behavior into a single model by applying Markov chains [16] and Petri nets [17]. However, the corresponding models are generally large since the state model of this model approximately represents the cross-product of the state-spaces of the availability and performance models, which is dealt with by using approximation techniques, including truncation, state lumping, and model composition [7] or by using automatic generation methods for Markov chains [18]. Furthermore, this model is considered stiff, since the performance related rates, including job arrival rates, are significantly larger than the failure related rates. The stiffness problem is tolerated by applying either aggregation techniques [19], [20] or stiffness-tolerant models [21].

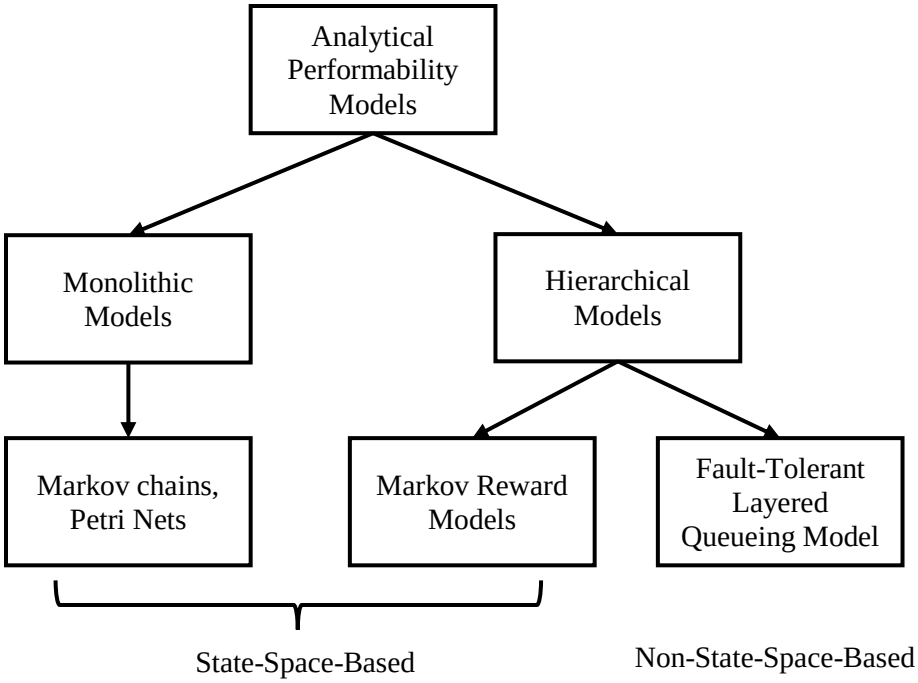

Figure 2.1 Analytical performability modeling approaches

Alternatively, the hierarchical approaches eliminate the largeness and stiffness problems associated to monolithic approach by composing the overall model of the system from a set of reduced non-stiff models. In this technique, since the performance related rates are several orders of magnitude larger than the failure related rates, it is assumed that the system reaches steady-state with regard to the performance related events between consecutive occurrence of failure events. The performance measures of the system are computed at each of these steady-states. Furthermore, the general system is categorized by weighing steady-state performance by applying structure state probabilities, which leads to a hierarchy of models: a higher-level structure state availability model representing the failure behavior of components and a number of lower level performance reward models, for every structural state in the availability model. The

comparison between the hierarchical models and monolithic models for a M/M/C/C queuing system in a wireless communication network is represented in [22].

Additionally, hierarchical models are categorized into state-space-based, i.e. Markov Reward Models [23]–[25] , and non-state-space-based models, i.e. Fault-Tolerant Layered Queuing Model [26], [27].

Analytical modeling approaches offer cost effective tools for analyzing cloud service performance, allowing the assessment of the influence of a large parameter space on service performance. A comparison between analytical modeling approaches for cloud service performance evaluation is provided in [11]. The queuing theory is a classical approach for modeling and evaluation of computer systems and has been significantly implemented for evaluating cloud service performance, including network calculus, which is an extension of the queuing theory. Alternatively, Stochastic Reward Net (SRN), an extension of Stochastic Petri Net (SPN), is exploited for modeling cloud service provisioning and service performance analysis. Xiong and Perros [28] modeled a cloud service provisioning system by implementing a queuing network incorporating two tandem servers with finite buffer servers and modeling each server as a classical M/M/1 queue. Goswami et al. [29] developed a model for cloud performance analysis by employing the virtualization feature. The proposed model is a M/M/m/N queue with *m* servers and a finite buffer of size *N*. Ellens et al. [30] developed an M/M/m/m queueing model with m servers and no buffers before the server for cloud computing centers with multiple priority classes. In order to simplify the modeling and assessment, the above techniques assumed exponentially distributed service time and/or inter-arrival time. This

assumption does not accurately demonstrate the realistic service feature of cloud infrastructures.

2.2.2. Simulation-Model Based Techniques

Computer simulations are an alternative to conventional analytical approaches. The main advantage of the simulation approaches over the corresponding analytical counterparts is flexibility in representation of complex systems at desired level of abstractions as well as low storage requirements. Simulation tools provide the opportunity of evaluating the hypothesis in a controlled environment, allowing fast and reliable reproduction of results. In general, simulation approaches provide substantial benefits including: evaluating services in a repeatable and controllable environment and adjustment of the system bottlenecks before deploying on real clouds. Several cloud simulators are developed to facilitate the performance analysis of cloud services and applications. SimGrid [31] is a general framework for simulation of distributed applications on Grid platforms. Additionally, GangSim is a Grid simulation toolkit that offers support for modeling of Grid-based virtual organizations and resources. Furthermore, GridSim [32] is an event-driven simulation toolkit for application in heterogeneous Grid resources which supports inclusive modeling of grid entities, users, machines, and network, including network traffic. The main drawback of Grid based simulation frameworks is their inability to support modeling of virtualization-enabled resource and application management environments. Cloudsim [33]–[35] is a novel and generalized java-based event-driven simulation framework that allows modeling and simulation of cloud computing infrastructures and services. The main advantages of the

application of Cloudsim are time effectiveness and flexibility and applicability. SimPy [36], [37] is a process-oriented discrete event simulation framework based on Python that provides fast and reliable tools for modeling and simulating cloud computing services and applications.

There are several different types of applications that go through a transition during a given time. Simulating these applications are mostly distinguished by the fact that the events are modeled either continuously or discretely. Consequently, there are two major simulation techniques for modeling dynamic events, namely continuous and discrete event simulations. Discrete event simulation [38] is a superior tool for modeling sophisticated system dynamics and stochastic processes. In contrary to the continuous event simulation, where events are simulated based on the changes through the given time frame with no interruption in between events, in discrete event simulation (DES) approximations are used to jump through the breaks that exists between the two given events, to the next step of the discrete sequence of events in time. The most important aspect of DES is the set of activities that happen in a given fraction of time. DESs are considered activity based simulation in contrary to the continuous event based simulations. Although, more resources may be needed for discrete event simulations, they generally execute quicker relative to continuous simulation, since they do not have to simulate all the time fractions and they only need the initial and ending point of any system. Discrete event simulation can also be used to predict or observe the behavior of a system under given circumstances. Simpy and Cloudsim are discrete event simulation

frameworks. This work has used Simpy framework for simulating the performance behavior of an application in presence of failures.

3. The Proposed Model

This chapter describes the proposed modeling technique. First, it described the model that is considered for a cloud-based application. Second, it describes a model for representing a fault management architecture of an application. Finally, it describes how to compute the performability measures. The proposed model accounts for the failure information propagation through the fault management architecture.

3.1. Application Model

An application provides certain kind of service to its end users. An application is an example of a SaaS software (Software-as-a-Service) owned by an application provider who elected to deploy it in a cloud. An application deployment in a cloud is composed of *m* application instances (*A1*, *A2*, …, *Am*). Each application instance runs on its own VM. For the modeling purposes in this work, it is assumed that all the VM instances are of the same type in terms of their processing speed. A similar model for an application has been assumed earlier by Calheiros et al. [39].

Figure 3.1 represents the application model as a network of queues. Each queue corresponds to an application instance hosted on its own VM. An application instance can fail and can be restarted. To restart a failed application instance, the failure needs to be detected first. The application model incorporates a Load Balancer (LB), which represents an infinite server and it is assumed to be failure-free. Let λ denote the arrival rate of jobs, the number of jobs per unit time received by the load balancer (LB).

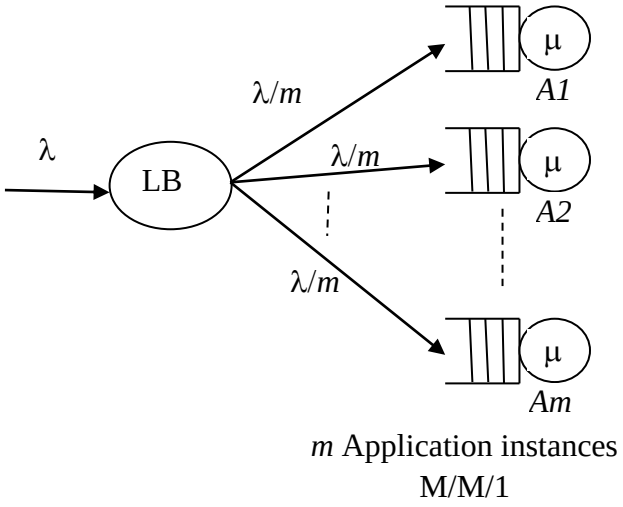

Figure 3.1 Application Model (as network of queues) where the load balancer (LB) *believes* that *m* application instances are operational.

The LB acts as a reverse proxy, which distributes the application traffic across the corresponding servers. The main benefits of LB includes increasing the capacity and reliability of applications, improving the overall application performance, and decreasing the work-load on servers associated with managing and maintaining applications. In this work, the LB distributes the workload equally among the application instances that it *believes* are operational. The phrase "believes are operational" here implies that LB has an impression that the application instances are operational, however, in reality, some instances might be in failed state. This belief of LB regarding an application instance depends on the fault management architecture and the status of the management components at the time of failure (or at the time of restart completion) of the instance. If LB believes that *m* application instances are operational, then the arrival rate at each of those instances will be λ/m. Let μ be the service rate of each application instance. The

application instances are represented by M/M/1 queues, where arrival times are determined by a Poisson process and job service times are represented by an exponential distribution. Furthermore, the servers process the client requests on a first-come, first-served (FIFO) order. Each server is associated with an infinite buffer (i.e. waiting area).

3.2. Fault Management Architecture Model

A separate fault management system [6] can be utilized to monitor the health of the application instances. The management system can detect and isolate a failure and can trigger actions such as automatic restart of a failed instance. It can also notify the load balancer concerning the status of the application instances, which in turn re-distributes the workload accordingly.

Failures of instances can be detected by mechanisms such as heartbeats and timeouts on periodic polls. Heartbeat messages from an application instance can be generated by a special heartbeat interrupt service routine which sends a message to one or more managers, every time an interrupt occurs, as long as the instance has not crashed. If an instance cannot initiate heartbeat messages, then it may be able to respond to messages from the manager(s); these are considered as the status polls. The responses provide the same information as heartbeat messages. Once the heartbeat information is collected, it can be propagated to other managers and finally to the load balancer.

The fault management architecture model described here has three types of components: application instances, managers and the load balancer. There are two types of arcs: *detect* (solid-line open-ended arrow) and *notify* (dashed-line open-ended arrow).

These arcs are typed according to the information they convey, in a way which supports the analysis of *belief* of the status of the application instances at different points in the management system. A *detect* arc from component *a* to component *b* convey data that component *b* can detect the crash failure of component *a* and can trigger automatic restart of component *a*. The time to detect the failure is assigned as a weight on the arc. A *notify* arc from component *a* to component *b* implies that component *a* propagates status data about application instance(s) that it has collected or received to component *b*. It is assumed that the notification happens in no time.

Upon occurrence of a failure of an application instance, the occurrence is first captured by the manager(s) monitoring that instance through a *detect* arc. Then the instance is restarted by the manager(s) and the failure information propagates through *notify* arcs, to other manager(s) and finally to the LB which initiate system reconfiguration by re-distributing the workload among the application instances it *believes* are operational. Once the instance has restarted, it can notify its manager(s) about its status in no time which in turn can propagate the status of the instance to other managers and the load balancer.

A manager can also monitor other managers. Upon occurrence of a failure of a manager, the occurrence is first captured by the manager(s) monitoring the failed manager through a *detect* arc. Then the instance is restarted by the monitoring manager(s). In this case, the information does not have to be propagated to the LB.

Figure 3.2 shows a centralized fault management architecture for the application model of Figure 3.1. *M1* has been introduced here as the central manager that monitors all the application instances (*A1, A2, …, Am*). *M1* notifies LB about the status of the application instances. An application instance *Ai* where $i = 1, 2, \ldots m$ fails with rate f_{Ai} and can be restarted with mean restart time $1/r_{Ai}$. The failure and restart rate of the application instances may be different if the different application instances were created using N-version programming [40]. The failure and restart rate of *M1* is assumed to be f_{M1} and r_{M1}. In Figure 3.2, a rectangle represents a component that contains its name, its failure rate and its restart rate. The LB is assumed to be failure-free. The weight of a *detect* arc (i, j), $d(i, j)$, is the time to detect the failure of component *i* by component *j*. In this model, since *M1* is not monitored by any other manager, its failure has to be detected and it has to be restarted by a human administrator. This situation can be modeled by a detect arc from M1 to LB with weight equal to the time to detect the failure of M1 by a human being. Usually this weight will be large as compared to the weights on the other detect arcs (that represent automatic detection without human intervention).

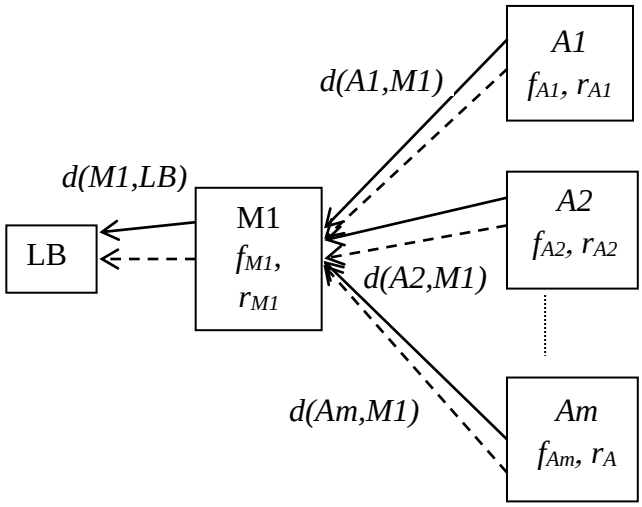

Figure 3.2 Centralized Fault Management Architecture Model for the application model of Figure 3.1. M1 is the central manager.

3.3. Modeling of Performability

The fault management architecture impacts the successful recovery of an application since the component responsible for reconfiguration (i.e. LB) might believe that an application instance is operational, however, in reality it may not be true. This might lead to job loss. This section describes how to compute the mean throughput of an application in presence of such job loss.

3.3.1. Failure Modeling of Components

Each component (either an application instance or a manager) except LB has three states: UP, FND (Failure-Not-Detected), FDR (Failure-Detected-and-Restarted).

Let ($M1_i$, $M2_i$,…, Ms_i) be the *operational* managers who are monitoring the health of component *i*. Since the failure is usually detected using heartbeat messages or

timeouts on periodic polls, the time to detect the failure of component *i* by manager Mk_i, $d(i, Mk_i)$, is deterministically distributed variable. Let δ be the minimum of the timespans taken by the managers $M1_i$, $M2_i$,…, Ms_i to detect the failure of component *i*, then, $\delta = \min \{ d(i, Mk_i)$ where k = 1, 2, …s$\}$.

The state transition diagram of a component *i* is shown in Figure 3.3. Each component *i* starts in the UP state. Once it fails with rate f_i, it transits to state FND (Failure-Not-Detected). The failure can be detected with rate $1/\delta$ at which point it is restarted. This is subject to the condition that at least one of the managers monitoring the component *i* is *operational* when component *i* fails. Otherwise, the component *i* will remain in FND state until one of its managers is operational again. Once the failure of component *i* is detected and it is restarted, it transits to state FDR (Failure-Detected-and-Restarted). The restart rate is r_i. Once the restart is complete, the component transits back to the UP state.

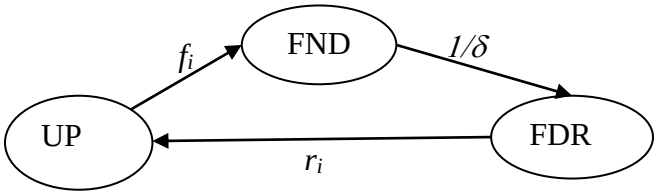

Figure 3.3 Failure Model of each component.

An application instance can process jobs only when it is in UP state. Similarly, a manager can monitor other components only when it is in UP state.

3.3.2. Modeling Failure Information Propagation through the fault management architecture

The load-balancer LB maintains a list containing the states of all the application instances. Each of these states can be either UP or FDR. These states of the application instances are the states that the LB *believes* to be true. LB distributes the workload equally among the application instances that it *believes* are in UP state.

Subsequently two situations and their consequences are described next where the state of *Ai* that LB *believes* to be true, becomes inconsistent with the actual state of *Ai*.

Situation 1- Assuming that LB contains the state of application instance *Ai* as UP. When the application instance *Ai* fails, then the state of *Ai* that LB has becomes inconsistent with the actual state of *Ai* (which is FND). Defining the time τ as follows:

Let *P* be the set of *operational paths* (i.e. paths containing all operational managers) from *Ai* to LB at the time of failure of *Ai*. The initial arc of each of these paths should be a *detect* arc and the rest should be *notify* arcs. Subsequently:

$$\tau = \min_{p \in P} \{ weight\ of\ \ the\ initial\ detect\ arc\ of\ \ path\ \ p \} \tag{3.1}$$

Here, jobs will be lost for duration τ since LB will continue sending the jobs to *Ai* although *Ai* has already failed.

If *P* is null, i.e. no operational path exists from *Ai* to LB at the time of failure of *Ai*, then LB continues to *believe* that *Ai* is UP although *Ai* has already failed. The jobs are lost until at least one such path exists or *Ai* comes back to UP state again.

Thus, job loss entirely depends on the fault management architecture, status of the managers at the time of failure, and the failure detection times.

Since job loss affects the mean throughput of the application, the state inconsistency of LB in this case decreases the mean throughput of the application.

Situation 2 - Assuming that LB contains the state of application instance *Ai* as FDR. When the application instance *Ai* restarts and comes back to UP state, then the state of *Ai* that LB has becomes inconsistent with the actual state of *Ai* (which is UP).

Let *P* be the set of *operational paths* (i.e. paths containing all operational managers) from *Ai* to LB at the time when *Ai* became operational. All the arcs of each of these paths should be *notify* arcs.

If *P* contains at least one *operational path*, then the state of *Ai* that LB has becomes consistent with the actual state of *Ai*.

If *P* is null, then LB *believes* that *Ai* is failed although *Ai* is in UP state. As a result, LB does not send any job to *Ai*. This will result in higher response time for jobs (that are not lost) compared to the case where the jobs were also sent to *Ai*.

The state inconsistency of LB in this case results in higher mean response time of the jobs that are not lost.

3.3.3. Computing Performability Measures

The application model and its fault management architecture model are simulated using a discrete event simulation framework called SimPy [36], [37]—a Python based framework.

Assuming that the application instances do not fail, then the mean throughput of the application will be same as the arrival rate, λ. However, if the failures of application instances and the managers are considered, some jobs will be lost. Let, NTL denote the *normalized throughput loss.* As per [7], NTL is the fraction of jobs that are lost. Let N denote the total number of job arrivals in one simulation run. At the end of each simulation run, the number of jobs that are lost, *n*, are computed. Subsequently, the *normalized throughput loss* is defined as:

$$NTL = \frac{n}{N} \tag{3.2}$$

Additionally, the mean throughput of the application, MeanTHR, can be computed as follows:

$$MeanTHR = \lambda\left(1 - NTL\right) \tag{3.3}$$

During each simulation run, the response time, RT_j, of each job *j* that is not lost, is recorded. Then, the mean response time, MeanRT, can be estimated as:

$$MeanRT = \sum_{j=1}^{N-n} \frac{RT_j}{N-n} \tag{3.4}$$

4. A Comparison of Five Fault Management Architectures

This chapter studies the effect of five different fault management architectures on the performance of an application in presence of failures. The effect of the fault management architectures on the mean throughput of the application and mean response time of a job that is not lost is evaluated.

The application model is represented in Figure 4.1 with three application instances (A1, A2 and A3), i.e. $m = 3$. It is assumed that the application provider requests to utilize three managers, M1, M2 and M3, for monitoring its application. Table 4.1 represents the model parameters and their values.

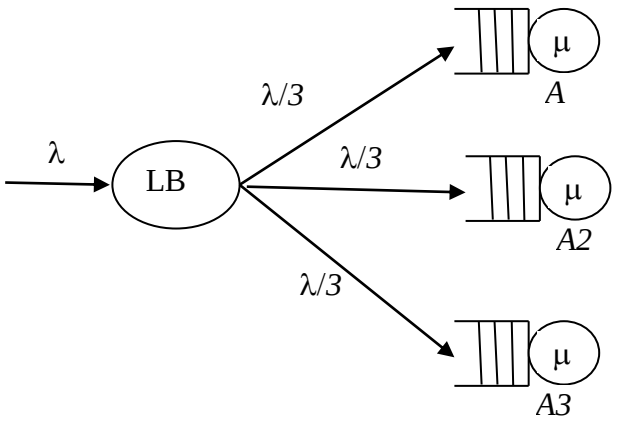

Figure 4.1 Application Model (as network of queues)

Table 4.1 Model Parameters

Parameter	**Parameter Value**
Mean inter-arrival time, $1/\lambda$	1 second
Mean Service time, τ	2 seconds
Mean Time to failure, $1/f_i$, where i = A1, A2, A3, M1, M2, M3	600 seconds
Mean time to restart, $1/r_i$, where i = A1, A2, A3, M1, M2, M3	60 seconds
Time to detect a failure automatically by a manager	120 seconds
Time to detect a failure by a human administrator	900 seconds

In this section, the focus is to attempt to answer the question "Which architecture to choose from the given five fault management architectures (each containing the same three managers) that will meet throughput and response time objectives"?

In the simulations, the mean inter-arrival time of jobs, $1/\lambda$, is assumed to be 1 second. The mean service time, τ, is assumed to be 2 seconds. The mean time to failure for each of the three application instances and for each of the three managers is assumed to be 10 minutes (=600 seconds), i.e. the failure rate is 0.00167 failures/sec. Similarly, the time to restart is assumed to be 60 seconds, i.e. the restart rate is 0.0167 restarts/sec. The load-balancer LB is assumed to be *failure-free*. The inter-arrival time, service time, time to failure, and time to restart are assumed to be exponentially distributed. The failure detection time is assumed to be deterministically distributed.

4.1. Description of the Fault Management Architectures

In this analysis, the following five fault management architectures are considered as given below:

Case-1: For this case it is assumed that, the application instance A1 is monitored by M1, A2 by M2, and A3 by M3. It is also assumed that only application instances can fail, and there is no management components failure for this case. Therefore, if an application instance fails, its manager will be able to detect that failure automatically in 120 seconds (2 minutes). The manager components cannot fail and the managers are not monitored by the other managers. Since the manager failures will not occur and all the three managers status is assumed to be always up in this case, there is no weight on the detect arc from each manager to LB, i.e. *d*(M1, LB), *d*(M2, LB), and *d*(M3, LB). The fault management architecture is shown in Figure 4.2.

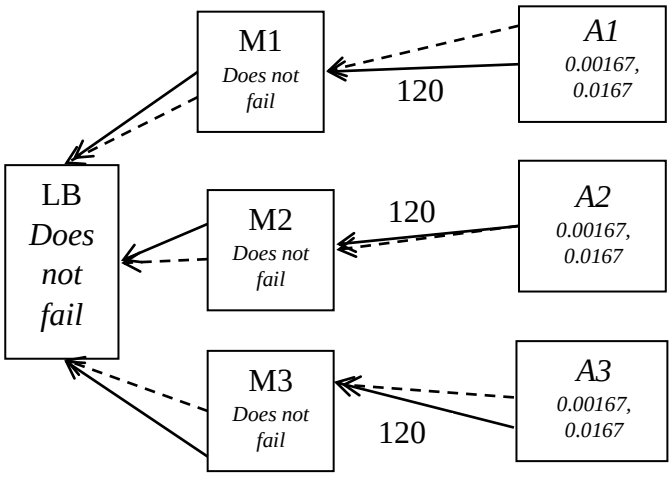

Figure 4.2 Case-1: Manager M1 monitors A1, M2 monitors A2, and M3 monitors A3. The manager components cannot fail.

Case-2: The application instance A1 is monitored by M1, A2 by M2, and A3 by M3. If an application instance fails, its manager will be able to detect that failure automatically in 2 minutes. The managers are not monitored by other managers. It is

assumed that the failure of managers has to be detected by human administrators. Since the manager failures are not automatically detected, the weight on the detect arc from each manager to LB is assumed to be 15 minutes, i.e. *d*(M1, LB), *d*(M2, LB), and *d*(M3, LB) = 900 seconds. The fault management architecture is shown in Figure 4.3.

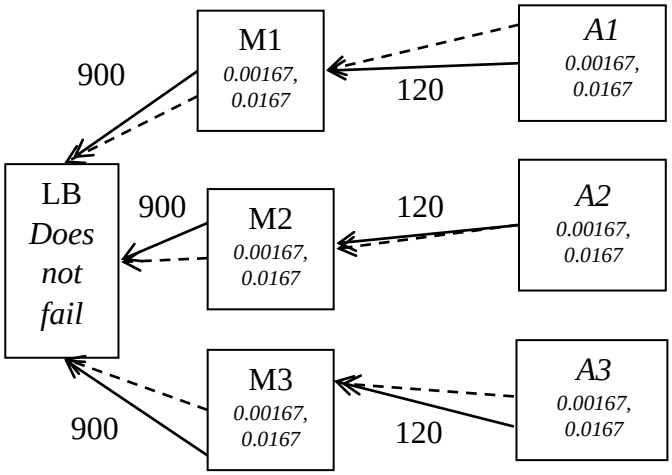

Figure 4.3 Case-2: Manager M1 monitors A1, M2 monitors A2, M3 monitors A3. The failure of managers has to be detected by human administrators.

Case-3: This case resembles a hierarchical management architecture. Application instances, A1 and A2, are monitored by manager M1 whereas A3 is monitored by manager M2. Managers, M1 and M2, are monitored by another manager M3. Here, it is assumed that the failure of M3 is not automatically detected. So, the weight on the detect arc from M3 to LB is assumed to be 15 minutes, i.e. *d*(M3, LB) = 900 seconds. The fault management architecture is shown in Figure 4.4.

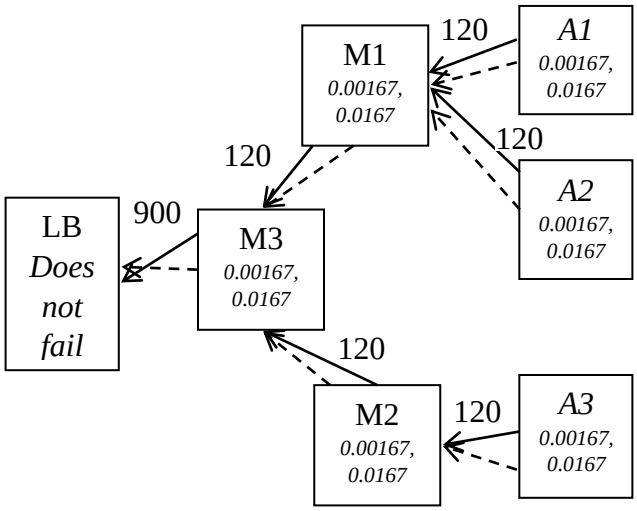

Figure 4.4 Case-3: M1 monitors A1 and A2, M2 monitors A3, M1 monitors M1 and M2. Failure of M3 has to be detected by human administrators.

Case-4: The application instance A1 is monitored by M1, A2 by M2, and A3 by M3 respectively. The managers also monitor each other. In this case, if the other two managers have already failed when the third one fails, that failure has to be detected by human administrators. The fault management architecture is shown in Figure 4.5. The weight of all the detect arcs that are not labeled in Figure 4.5 is assumed to be 120.

Case-5: Each application instance is monitored by all the three managers. The managers also monitor each other. In case, if the other two managers have already failed when the third one fails, that failure has to be detected by human administrators. The fault management architecture is shown in Figure 4.6. The weight of all the detect arcs that are not labeled in Figure 4.6 is assumed to be 120. This architecture has the highest number of detection paths from an application instance to the load balancer LB.

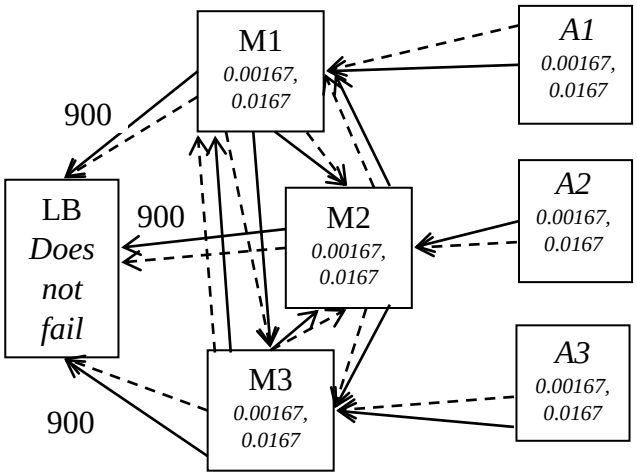

Figure 4.5 Case-4: The weight of all the detect arcs that are not labeled is 120. Managers monitor each other.

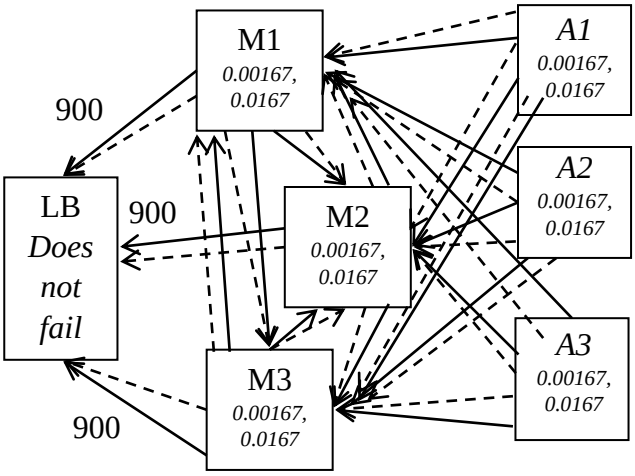

Figure 4.6 Case-5: The weight of all the detect arcs that are not labeled is 120. Each application instance is monitored by all the three managers. Managers monitor each other.

The application model and the corresponding fault management architecture model are simulated using SimPy 3.0.8. The simulation time was 9000 seconds with 10 simulation runs for each case. Since the number of simulation runs is < 30, the confidence interval formula that involves t-distribution is used rather than standard normal distribution. Since the number of simulation runs is 10, the degrees of freedom for t-distribution is equal to 9. The t value for 95% confidence with degree of freedom equal to 9 is t = 2.262. The 95% confidence interval for the mean response time is (MeanRT $\pm$ 2.262*SEM) where SEM is the standard error of mean. Similarly, the confidence interval for the mean throughput and the NTL are computed. The error bars are shown in each of the figures given below.

4.2. The Effect of Varying the Mean Inter-Arrival Time

Table 4.2. demonstrates the results of the simulations for the effect of the mean inter-arrival time on the values of the normalized throughput loss, mean throughput, and mean response time of jobs that are not lost. In the simulations, the mean inter-arrival time (i.e. $1/\lambda$) varies from 0.7 second to 1.5 seconds while maintaining the other parameter values same as Table 4.1. Figure 4.7 demonstrates the Table 4.2 simulation data. According to Figure 4.7(a), for high workload (i.e. for mean inter-arrival time 0.7 second), Case-5 has lower NTL (i.e. higher mean throughput) as compared to the other four cases. This is because Case-5 is densely connected and it has more number of detection paths from the application instances to LB through the fault management architecture. However, based on Figure 4.7(b), Case-5 has higher MeanRT for jobs that are not lost. This is because in Case-5, more number of jobs is accepted by the

application and thus, the number of jobs competing for resources is higher than the other three cases.

Furthermore, for high workload (i.e. for mean inter-arrival time 0.7 second),

- If the application provider wants a management architecture that will give maximum throughput, then Case-5 will be the choice (i.e. Figure 4.7(c)). On the other hand, if the provider wants an architecture that will give minimum mean response time for jobs that are accepted by the application, then Case-2 will be the choice (i.e. Figure 4.7(b)).

- If the application provider has some performance objectives that has to be met by the application to its end-users, for example, average throughput greater or equal to 1job/sec and average response time of 45 seconds or less, then Case-4 will be the architecture of choice.

For low workload (i.e. for mean inter-arrival time 1.5 seconds), all the cases have almost same MeanRT, however, Case-5 has higher throughput and lower NTL compared to the other cases. Consequently, for low workload circumstances, Case-5 will be the choice.

Table 4.2 The simulation results for the effect of varying the mean inter-arrival time

1/λ	NTL Mean	NTL SEM	ArvRt Mean	ArvRt SEM	Resptm Mean	Resptm SEM	THR Mean	THR SEM
Case-1								
0.7	0.159	0.007	1.43	0.006	49.835	5.638	1.196	0.01
0.9	0.109	0.006	1.113	0.005	10.965	0.642	0.991	0.007
1.1	0.116	0.006	0.911	0.004	6.176	0.44	0.805	0.005
1.3	0.101	0.007	0.777	0.002	4.576	0.405	0.698	0.005
1.5	0.091	0.005	0.67	0.003	3.51	0.158	0.609	0.005
Case-2								
0.7	0.402	0.015	1.43	0.006	34.296	1.94	0.852	0.021
0.9	0.359	0.019	1.115	0.004	9.051	0.615	0.715	0.022
1.1	0.375	0.028	0.913	0.004	5.105	0.42	0.57	0.025
1.3	0.333	0.017	0.775	0.002	4.167	0.32	0.517	0.014
1.5	0.373	0.021	0.673	0.002	3.326	0.216	0.422	0.014
Case-3								
0.7	0.364	0.026	1.434	0.006	42.498	4.753	0.909	0.038
0.9	0.338	0.018	1.119	0.004	11.187	0.58	0.74	0.02
1.1	0.318	0.03	0.913	0.005	5.083	0.294	0.623	0.029
1.3	0.337	0.02	0.774	0.002	4.065	0.214	0.513	0.016
1.5	0.329	0.023	0.671	0.002	3.36	0.127	0.45	0.015
Case-4								
0.7	0.231	0.009	1.429	0.006	42.437	2.49	1.094	0.016
0.9	0.162	0.011	1.114	0.005	10.828	0.788	0.931	0.012
1.1	0.17	0.008	0.913	0.004	5.202	0.264	0.757	0.009
1.3	0.164	0.007	0.772	0.003	3.928	0.174	0.645	0.005
1.5	0.164	0.018	0.676	0.003	3.383	0.115	0.565	0.012
Case-5								
0.7	0.157	0.012	1.434	0.006	47.676	2.794	1.203	0.015
0.9	0.086	0.006	1.115	0.004	12.079	0.855	1.018	0.008
1.1	0.099	0.008	0.911	0.004	5.913	0.267	0.82	0.005
1.3	0.095	0.013	0.774	0.003	4.548	0.274	0.7	0.012
1.5	0.077	0.003	0.673	0.003	3.371	0.101	0.621	0.003

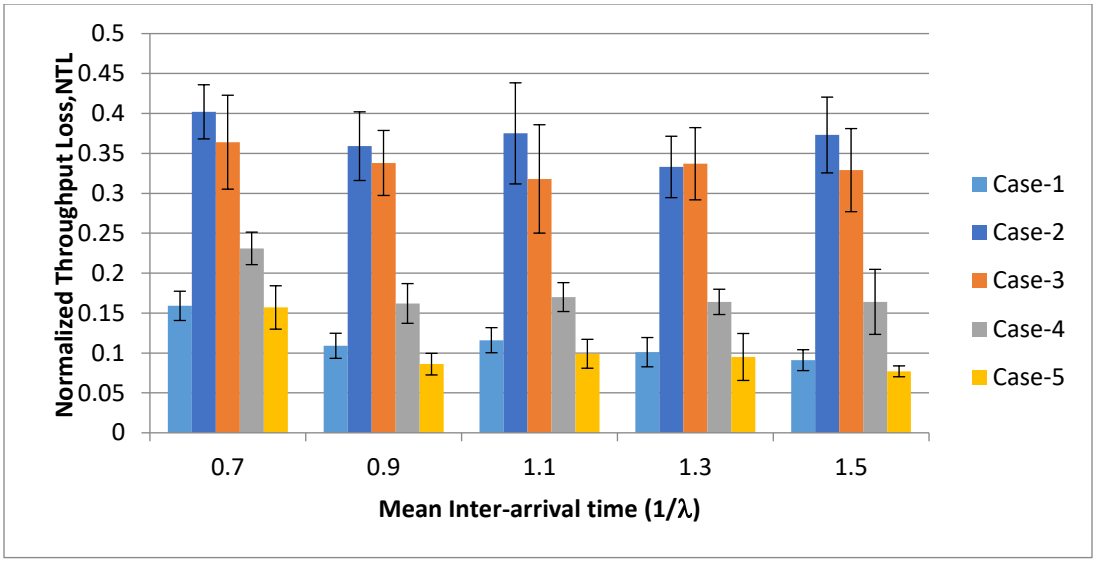

Figure 4.7(a)

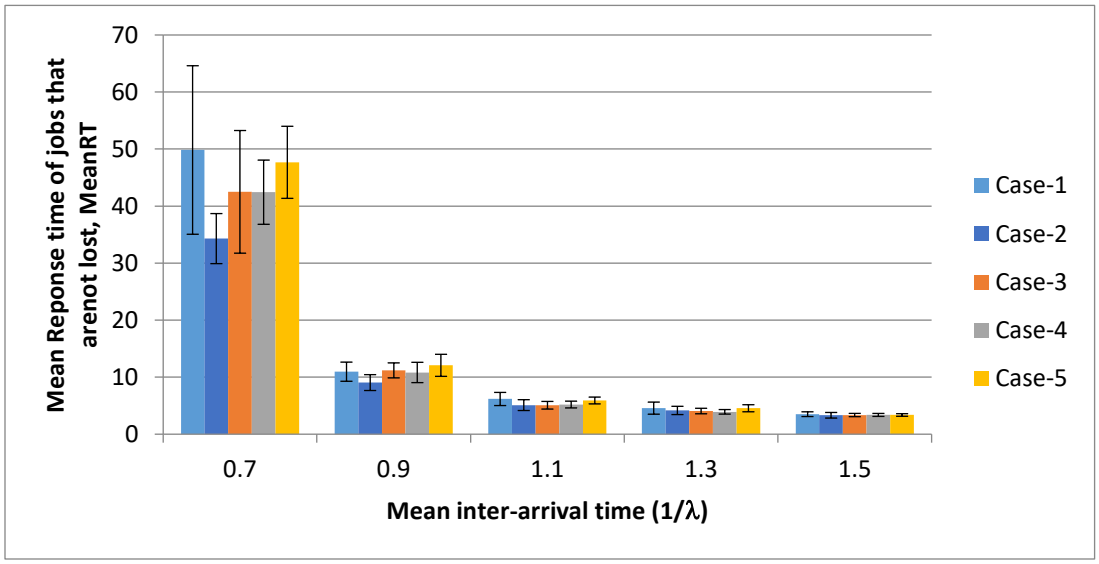

Figure 4.7(b)

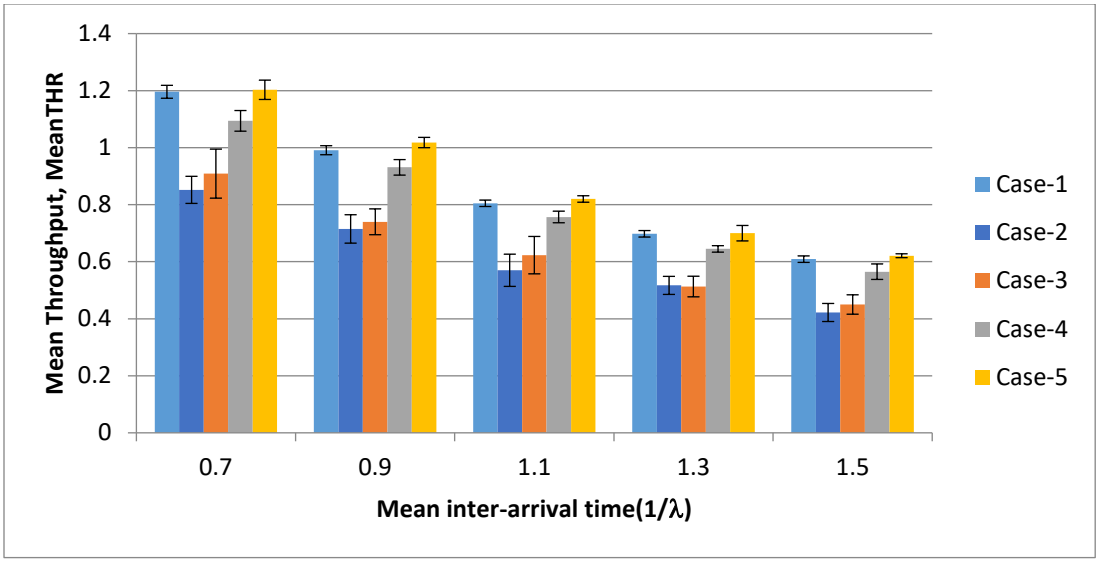

Figure 4.7(c)

Figure 4.7(a) Normalized throughput loss (NTL), (b) Mean response time of jobs that are not lost, and (c) Mean throughput. This is shown for all five cases, Case-1 to Case-5 (with varying mean inter-arrival time).

4.3. The Effect of Varying the Mean Time to Failure

Table 4.3 incorporates the results of the simulations for the effect of varying the mean failure time on the normalized throughput loss (NTL), mean throughput, and mean response time of jobs that are not lost .Subsequently, Figure 4.8 demonstrates the simulation results in Table 4.3 for the values of the normalized throughput loss (NTL), mean throughput, and mean response time of jobs that are not lost with the mean time to failure of the three managers (i.e. $1/f_{M1}$, $1/f_{M2}$, $1/f_{M3}$) varying from 600 seconds to 1800 seconds, while maintaining the other parameter values same as Table 4.1.

According to Figure 4.8, as the time to failure for the three managers increases, NTL decreases (Figure 4.8(a)). Consequently, the average throughput increases for all the

five cases (Figure 4.8(b)). The proportion of increase is higher for Case-2 as compared to the other cases. Subsequently, it is assumed that the throughput objective is 0.8 jobs/sec and response time objective to be met is 8 seconds or less. Figure 4.8 suggests that in order to meet both these objectives, it is required to buy the managers whose failure times are 20 minutes or more and select either Case-3 or Case-4. Since more reliable managers may be more expensive than less reliable ones, it will be cost effective to buy the managers whose failures times are 20 minutes and then select the architecture in Case-4, since Case-4 has similar MeanTHR and lower MeanRT than Case-3.

Table 4.3 The simulation results for the effect of varying the mean time to failure

Failure time	NTL Mean	NTL SEM	ArvRt Mean	ArvRt SEM	Resptm Mean	Resptm SEM	THR Mean	THR SEM
Case-1								
600	0.103	0.004	1.001	0.004	8.586	0.785	0.898	0.007
1200	0.103	0.004	1.001	0.004	8.586	0.785	0.898	0.007
1800	0.103	0.004	1.001	0.004	8.586	0.785	0.898	0.007
Case-2								
600	0.377	0.019	1.005	0.004	6.818	0.317	0.625	0.018
1200	0.255	0.016	1.006	0.003	6.659	0.447	0.749	0.015
1800	0.194	0.018	1.007	0.003	6.271	0.236	0.812	0.019
Case-3								
600	0.318	0.029	1.002	0.005	7.711	0.479	0.684	0.031
1200	0.149	0.011	1.006	0.004	7.97	0.662	0.856	0.014
1800	0.152	0.021	1.003	0.003	7.258	0.42	0.85	0.02
Case-4								
600	0.148	0.013	1.003	0.003	6.415	0.207	0.854	0.013
1200	0.145	0.01	1.003	0.005	7.449	0.464	0.857	0.01
1800	0.136	0.006	1.004	0.004	7.343	0.404	0.867	0.008
Case-5								
600	0.096	0.018	1.005	0.004	7.346	0.389	0.907	0.018
1200	0.087	0.004	1.005	0.004	8.223	0.956	0.917	0.005
1800	0.071	0.003	1.006	0.005	7.756	0.4	0.933	0.005

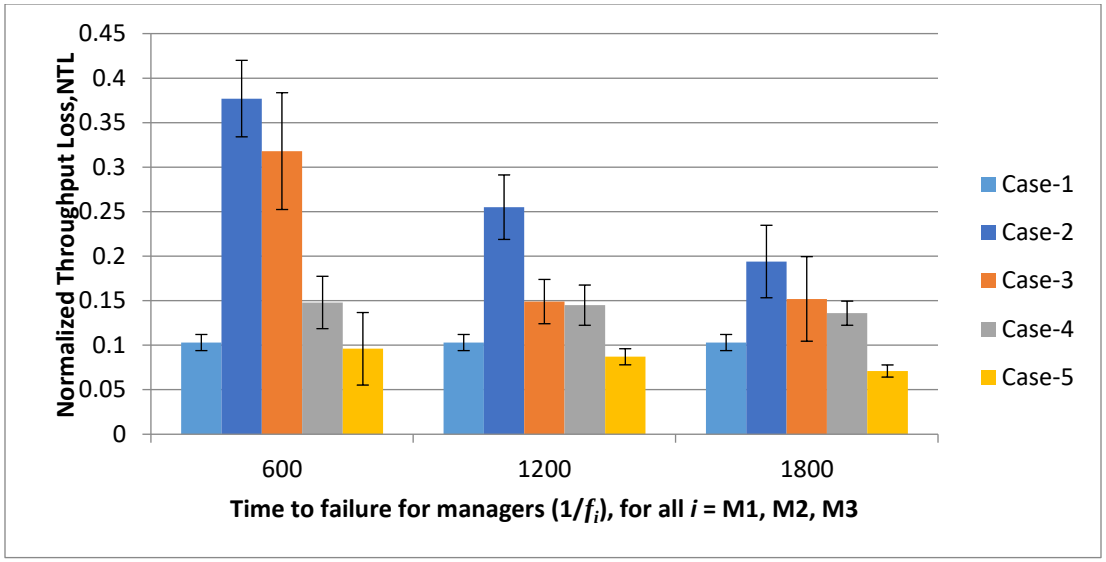

Figure 4.8(a)

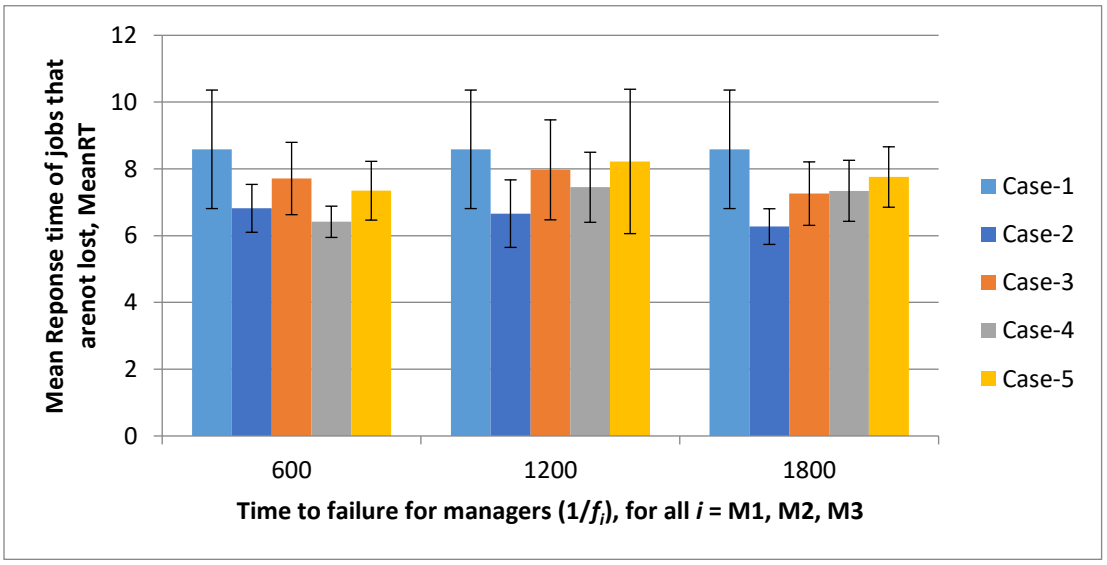

Figure 4.8(b)

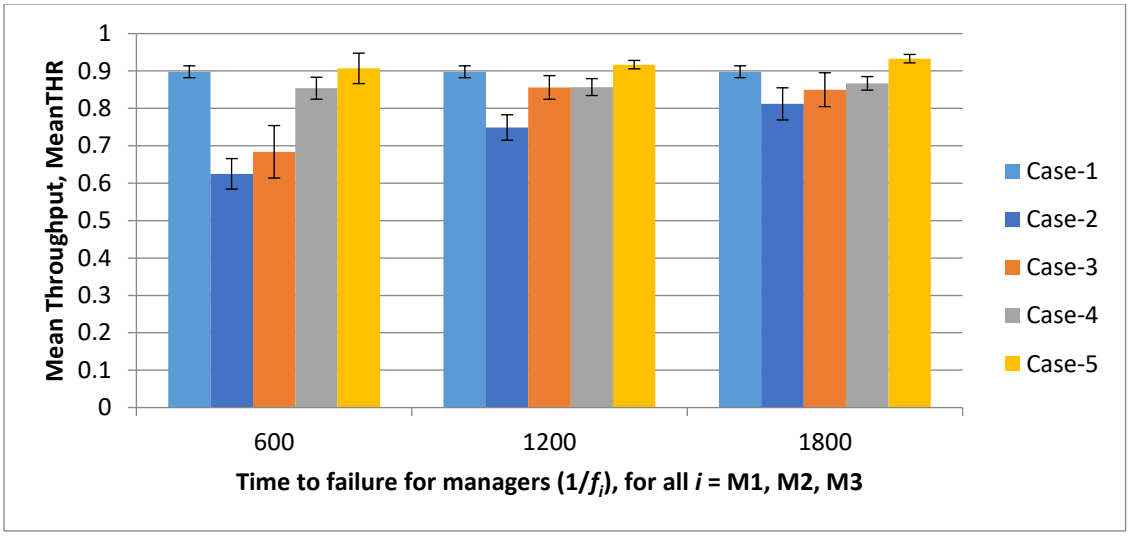

Figure 4.8(c)

Figure 4.8 (a) Normalized throughput loss (NTL), (b) Mean response time of jobs that are not lost, and (c) Mean throughput. This is shown for all five cases, Case-1 to Case-5 (with varying mean time to failure of the three managers).

4.4. The Effect of Varying the Failure Detection Time

This section discusses the values of the normalized throughput loss (NTL), mean throughput, and mean response time of jobs that are not lost with the failure detection time by the three managers varying from 60 seconds to 180 seconds while maintaining the other parameter values same as Table 4.1. The results of the SimPy simulations are represented in Table 4.4.

Figure 4.9 demonstrates the results of the simulations in Table 4.4. According to Figure 4.9, for the objective to maximize the system throughput, it will be better to choose 1-minute detection interval with Case-5 as the fault management architecture. On the other hand, if the goal is to minimize the response time of jobs that are not lost, then

for 1-minute detection interval, Case-2 will be the choice. For 2-minute detection interval, Case-3 is worse compared to Case-4 and Case-5, since it demonstrates lower throughput and higher response times of jobs that are not lost compared to the other two cases.

Table 4.4 The simulation results for the effect of varying the failure detection time

Detection Time	NTL Mean	NTL SEM	ArvRt Mean	ArvRt SEM	Resptm Mean	Resptm SEM	THR Mean	THR SEM
Case-1								
60	0.062	0.006	1.004	0.004	7.386	0.409	0.941	0.006
120	0.103	0.004	1.001	0.004	8.586	0.785	0.898	0.007
180	0.134	0.007	1.005	0.004	8.102	0.485	0.87	0.006
Case-2								
60	0.302	0.021	1.002	0.005	6.235	0.337	0.699	0.022
120	0.377	0.019	1.005	0.004	6.818	0.317	0.625	0.018
180	0.489	0.016	1.004	0.004	5.714	0.257	0.513	0.017
Case-3								
60	0.181	0.014	1.011	0.003	8.133	0.396	0.827	0.015
120	0.318	0.029	1.002	0.005	7.711	0.479	0.684	0.031
180	0.419	0.052	1.004	0.004	6.247	0.36	0.583	0.052
Case-4								
60	0.087	0.005	1.007	0.003	8.324	0.568	0.918	0.004
120	0.148	0.013	1.003	0.003	6.415	0.207	0.854	0.013
180	0.27	0.016	1.004	0.004	6.432	0.239	0.732	0.015
Case-5								
60	0.05	0.004	1.001	0.006	7.722	0.622	0.95	0.006
120	0.096	0.018	1.005	0.004	7.346	0.389	0.907	0.018
180	0.131	0.013	1.007	0.003	8.208	0.684	0.874	0.014

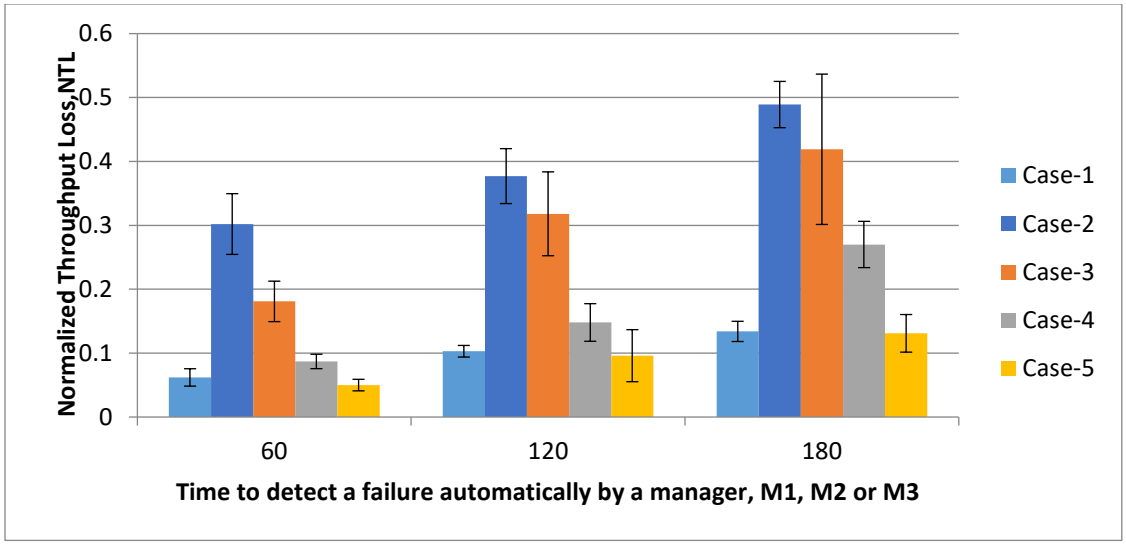

Figure 4.9(a)

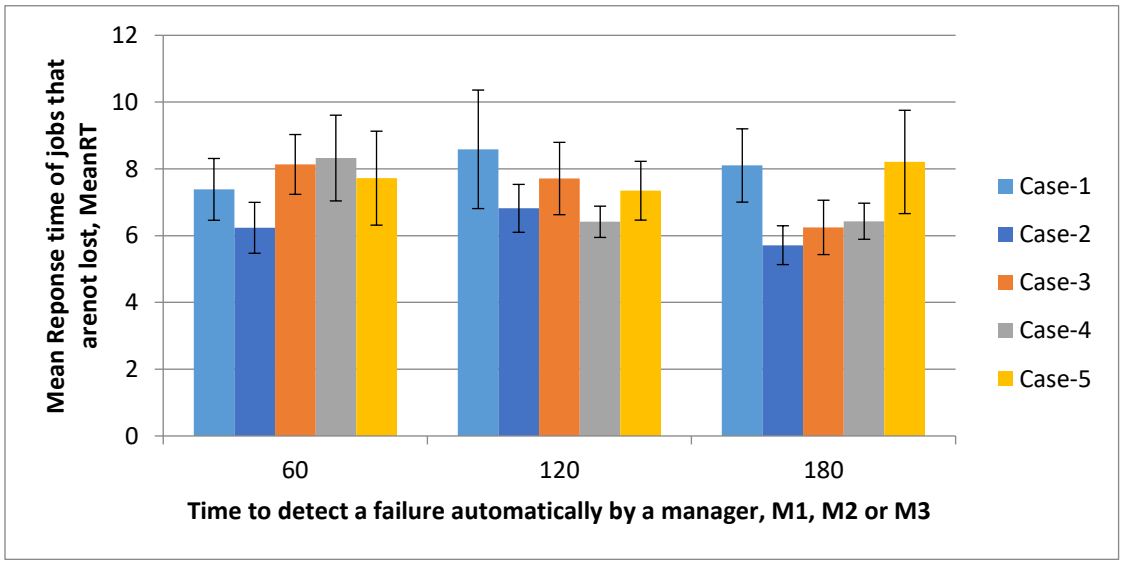

Figure 4.9(b)

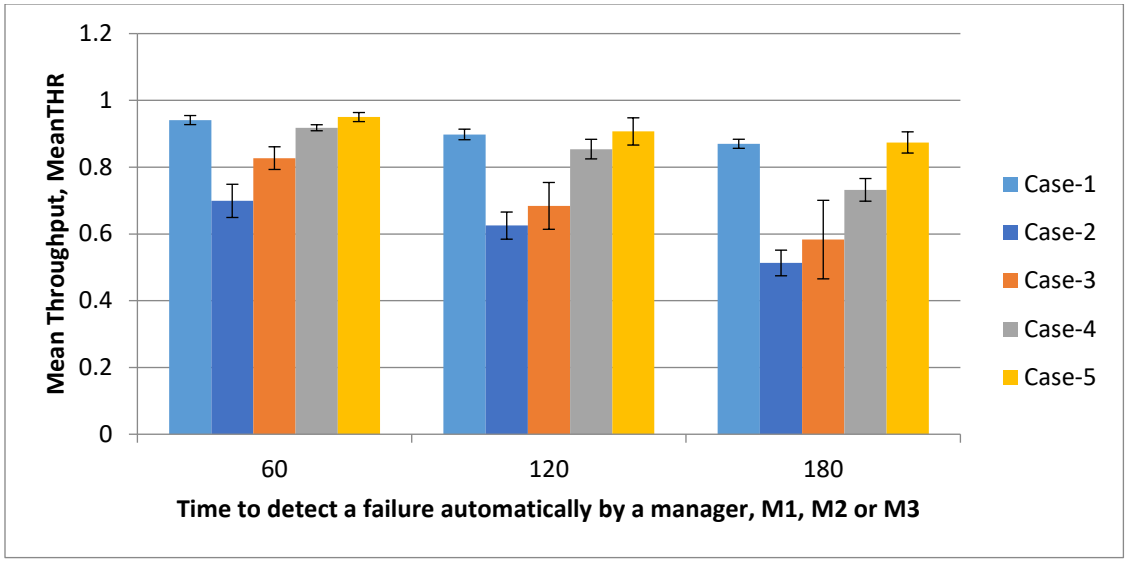

Figure 4.9(c)

Figure 4.9 (a) Normalized throughput loss (NTL), (b) Mean response time of jobs that are not lost, and (c) Mean throughput. This is shown for all five cases, Case-1 to Case-5 (with varying failure detection time by the three managers).

5. Conclusion

This thesis proposed a discrete event simulation model that accounts for the impact of the fault management-architecture on the performance of a cloud-based application. The value of including the fault management architecture in the analysis is first to account for the failures and restarts of managers, second is to include delays to detect the failures, and third is to evaluate the limitations of the fault management architecture. These three considerations increase the fraction of jobs that are lost thereby affecting the system throughput. In this thesis, first, the application model, then the fault management architecture model, and finally how to compute the performability measures were described. The application model incorporated a load balancer that evenly distributed the workload evenly among the operational application instances. A centralized fault management architecture model, including application instances, managers, and the load balancer, was assumed for the application model to monitor the health of the application instances. Subsequently, the application model and the corresponding fault management architecture model were simulated by using the SimPy discrete event simulation framework and the performability measures, including the normalized throughput loss, mean throughput of the application, and the mean response time were investigated.

This thesis demonstrated the application of the proposed modeling technique by comparing five different sample fault management architectures. The effect of varying

the mean inter-arrival time, time to failure for managers, and time to detect a failure automatically by a manager on the performance measures was computed and discussed.

The examples provided in this work considered only failures of application instances and managers. This is because, generally, the virtual machines and physical machines are highly available and they are typically managed by the cloud providers. However, if the application provider also requires monitoring the virtual machines and/or physical machines allocated for their applications, then that can be effortlessly incorporated into the analysis as well.

The proposed model will benefit application providers to answer several important questions related to selection of fault management architecture for their application, including the appropriate fault management architecture for monitoring the application instances to satisfy the throughput and response time SLAs as well as the proper detection interval required for configuring the managers.

The results gathered from the study of the five different sample fault management architectures, can be used in applications with more than three managers or three application instances, since the outcomes can be generalized to n managers and n application instances. Furthermore, other application architectures such as multi-tier architectures can be investigated in future works.

References

[1] D. A. Menascé and P. Ngo, "Understanding Cloud Computing: Experimentation and Capacity Planning," in *Computer Measurement Group Conference*, 2009.

[2] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic load balancing on Web-server systems," *IEEE Internet Comput.*, vol. 3, no. 3, pp. 28–39, 1999.

[3] N. Grozev and R. Buyya, "Multi-Cloud Provisioning and Load Distribution for Three-Tier Applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 3, pp. 1–21, Oct. 2014.

[4] K. Al Nuaimi, N. Mohamed, M. Al Nuaimi, and J. Al-Jaroodi, "A Survey of Load Balancing in Cloud Computing: Challenges and Algorithms," *2012 Second Symp. Netw. Cloud Comput. Appl.*, pp. 137–142, Dec. 2012.

[5] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, p. 45, Jan. 2011.

[6] "ManageEngine," 2017. [Online]. Available: https://www.manageengine.com/. [Accessed: 27-Mar-2017].

[7] K. S. Trivedi, J. K. Muppala, S. P. Woolet, and B. R. Haverkort, "Composite performance and dependability analysis," *Perform. Eval.*, vol. 14, pp. 197–215, 1992.

[8] G. Bolch, *Queueing networks and Markov chains : modeling and performance evaluation with computer science applications*. Wiley-Interscience, 2006.

[9] T. Erl, *Service-oriented Architecture: Concepts, Technology, and Desing*. Boston: Prentice Hall, 2005.

[10] G. Zamani and O. Das, "Impact of Fault Management Architecture on Performance of Cloud-based Applications," in *Submitted to the 13th European Dependable Computing Conference (EDCC 2017)*, 2017.

[11] Q. Duan, "Cloud service performance evaluation: status, challenges, and opportunities – a survey from the system modeling perspective," *Digit. Commun. Networks*, vol. 3, no. 2, pp. 101–111, 2017.

[12] V. Stantchev, "Performance Evaluation of Cloud Computing Offerings," *2009 Third Int. Conf. Adv. Eng. Comput. Appl. Sci.*, pp. 187–192, Oct. 2009.

[13] G. Ataş and V. C. Gungor, "Performance evaluation of cloud computing platforms using statistical methods," *Comput. Electr. Eng.*, vol. 40, no. 5, pp. 1636–1649, 2014.

[14] Meyer, “On Evaluating the Performability of Degradable Computing Systems,” *IEEE Trans. Comput.*, vol. C-29, no. 8, pp. 720–731, Aug. 1980.

[15] K. S. Trivedi, G. Ciardo, M. Malhotra, and R. A. Sahner, “Dependability and performability analysis,” *Perform. Eval. Comput. Commun. Syst.*, pp. 587–612, 1993.

[16] N. Privault, *Understanding Markov chains : examples and applications*. Springer, 2013.

[17] M. Diaz, *Petri nets : fundamental models, verification and applications*. ISTE, 2009.

[18] B. R. Haverkort and K. S. Trivedi, “Specification techniques for Markov reward models,” *Discret. Event Dyn. Syst. Theory Appl.*, vol. 3, no. 2–3, pp. 219–247, Jul. 1993.

[19] Bobbio and Trivedi, “An Aggregation Technique for the Transient Analysis of Stiff Markov Chains,” *IEEE Trans. Comput.*, vol. C-35, no. 9, pp. 803–814, Sep. 1986.

[20] A. Reibman, K. Trivedi, S. Kumar, and G. Ciardo, “Analysis of Stiff Markov Chains,” *ORSA J. Comput.*, vol. 1, no. 2, pp. 126–133, May 1989.

[21] M. Malhotra, J. K. Muppala, and K. S. Trivedi, “Stiffness-tolerant methods for transient analysis of stiff Markov chains,” *Microelectron. Reliab.*, vol. 34, no. 11, pp. 1825–1841, Nov. 1994.

[22] Yue Ma, J. J. Han, and K. S. Trivedi, “Composite performance and availability analysis of wireless communication networks,” *IEEE Trans. Veh. Technol.*, vol. 50, no. 5, pp. 1216–1223, 2001.

[23] R. Ghosh, K. S. Trivedi, V. K. Naik, and D. S. Kim, “End-to-End Performability Analysis for Infrastructure-as-a-Service Cloud: An Interacting Stochastic Models Approach,” *2010 IEEE 16th Pacific Rim Int. Symp. Dependable Comput.*, pp. 125–132, Dec. 2010.

[24] F. Longo, R. Ghosh, V. K. Naik, and K. S. Trivedi, “A scalable availability model for Infrastructure-as-a-Service cloud,” *2011 IEEE/IFIP 41st Int. Conf. Dependable Syst. Networks*, pp. 335–346, Jun. 2011.

[25] J. Dantas, R. Matos, J. Araujo, and P. Maciel, “An availability model for eucalyptus platform: An analysis of warm-standy replication mechanism,” *2012 IEEE Int. Conf. Syst. Man, Cybern.*, pp. 1664–1669, Oct. 2012.

[26] O. Das and C. Murray Woodside, “Evaluating layered distributed software systems with fault-tolerant features,” *Perform. Eval.*, vol. 45, no. 1, pp. 57–76, 2001.

[27] O. Das and C. Murray Woodside, “The fault-tolerant layered queueing network

model for performability of distributed systems," *Proceedings. IEEE Int. Comput. Perform. Dependability Symp. IPDS'98 (Cat. No.98TB100248)*, pp. 132–141, 1998.

[28] K. Xiong and H. Perros, "Service Performance and Analysis in Cloud Computing," *2009 Congr. Serv. - I*, pp. 693–700, Jul. 2009.

[29] V. Goswami, S. S. Patra, and G. B. Mund, "Performance analysis of cloud with queue-dependent virtual machines," *2012 1st Int. Conf. Recent Adv. Inf. Technol.*, pp. 357–362, Mar. 2012.

[30] W. Ellens, M. Ivkovic, J. Akkerboom, R. Litjens, and H. van den Berg, "Performance of Cloud Computing Centers with Multiple Priority Classes," *2012 IEEE Fifth Int. Conf. Cloud Comput.*, pp. 245–252, Jun. 2012.

[31] A. Legrand, L. Marchal, and H. Casanova, "Scheduling distributed applications: the SimGrid simulation framework," *CCGrid 2003. 3rd IEEE/ACM Int. Symp. Clust. Comput. Grid, 2003. Proceedings.*, pp. 138–145, 2003.

[32] R. Buyya and M. Murshed, "GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing," *Concurr. Comput. Pract. Exp.*, vol. 14, no. 13–15, pp. 1175–1220, Nov. 2002.

[33] R. N. Calheiros, R. Ranjan, C. A. F. De Rose, and R. Buyya, "CloudSim: A Novel Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services," Mar. 2009.

[34] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exp.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[35] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of scalable Cloud computing environments and the CloudSim toolkit: Challenges and opportunities," *2009 Int. Conf. High Perform. Comput. Simul.*, pp. 1–11, Jun. 2009.

[36] "Simpy," 2017. [Online]. Available: https://simpy.readthedocs.org/. [Accessed: 27-Mar-2017].

[37] V. Castillo, "Parallel Simulations of Manufacturing Processing using Simpy, a Python-Based Discrete Event Simulation Tool," *Proc. 2006 Winter Simul. Conf.*, pp. 2294–2294, Dec. 2006.

[38] B. Sharda and S. J. Bury, "Evaluating production improvement opportunities in a chemical plant: a case study using discrete event simulation," *J. Simul.*, vol. 6, no. 2, pp. 81–91, May 2012.

[39] R. N. Calheiros, R. Ranjan, and R. Buyya, "Virtual Machine Provisioning Based

on Analytical Performance and QoS in Cloud Computing Environments," *2011 Int. Conf. Parallel Process.*, pp. 295–304, Sep. 2011.

[40] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1491–1501, Dec. 1985.