Cost-Based Automatic Recovery Policy in Data Centers

by

Yi Luo

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

Declaration

Abstract

Today's data centers either provide critical applications to organizations or host computing clouds used by huge Internet populations. Their size and complex structure make management difficult, causing high operational cost. The large number of servers with various different hardware and software components cause frequent failures and need continuous recovery work. Much of the operational cost is from this recovery work. While there is significant research related to automatic recovery, from automatic error detection to different automatic recovery techniques, there is currently no automatic solution that can determine the exact fault, and hence the preferred recovery action. There is some study on how to automatically select a suitable recovery action without knowing the fault behind the error.

In this thesis we propose an estimated-total-cost model based on analysis of the cost and the recovery-action-success probability. Our recovery-action selection is based on minimal estimated-total-cost; we implement three policies to use this model under different considerations of failed recovery attempts. The preferred policy is to reduce the recovery-action-success probability when it failed to fix the error; we also study different reduction coefficients in this policy. To evaluate the various policies, we design and implement a simulation environment. Our simulation experiments demonstrate significant cost improvement over previous research based on simple heuristic models.

Acknowledgements

I would like to express my heartfelt thanks to all the people who provided kindly helps during my research. Especially, I am deeply grateful to my supervisor, Professor Paul Ward, for giving me this great research opportunity, also for his constant and kindly guidance, encouragement and helps. He always points out new directions when I encountered difficulties, encourages me when I felt frustrated. With his enthusiastic helps, my research becomes much smoother and this thesis can be finished successfully.

Also, I greatly appreciate Professor Sagar Naik and Professor Krzysztof Czarnecki to review my thesis and provide their opinions and corrections to me.

Last but not least, many thanks to my family - my beloved wife, my adorable kids and my parents. I particularly thank my wife, for her material and spiritual supports to help me go through the hard time, for her extreme patience to put up with me. Without their unlimited and unconditional supports and love, I would not be able to achieve my goal.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Computers, networks, and their provided services are the essential in our life today. The technology behind this has grown rapidly, from the mainframe with connected terminals, to the local-area network with a client/server environment, to the widely used browser and Internet services. The usage has moved from restricted scientific and military research to everything in our life, personal or business, government, etc.

1.1 Cloud Computing

As the web developed from the 70s, along with faster and more reliable networking, more and more information and services are provided over the Internet. People increasingly rely on these ubiquitous information and services. Web Services and Service-Oriented Architecture emerged to enable the infrastructure and framework to provide more functions and services through the network. The large number of users and their requirements continuously moved the development of the technology. Cloud computing recently appeared to provide more traditionally local services on the Internet. Cloud computing is defined as a Internet-based computing, whereby shared resources, software, and information are provided to computers and other devices on demand.[61] There are different types of cloud providing different level of services [23]:

- IaaS - Infrastructure as a Service
- PaaS - Platform as a Service
- SaaS - Software as a Service

The IaaS cloud provides computer-infrastructure resources as an Internet service to

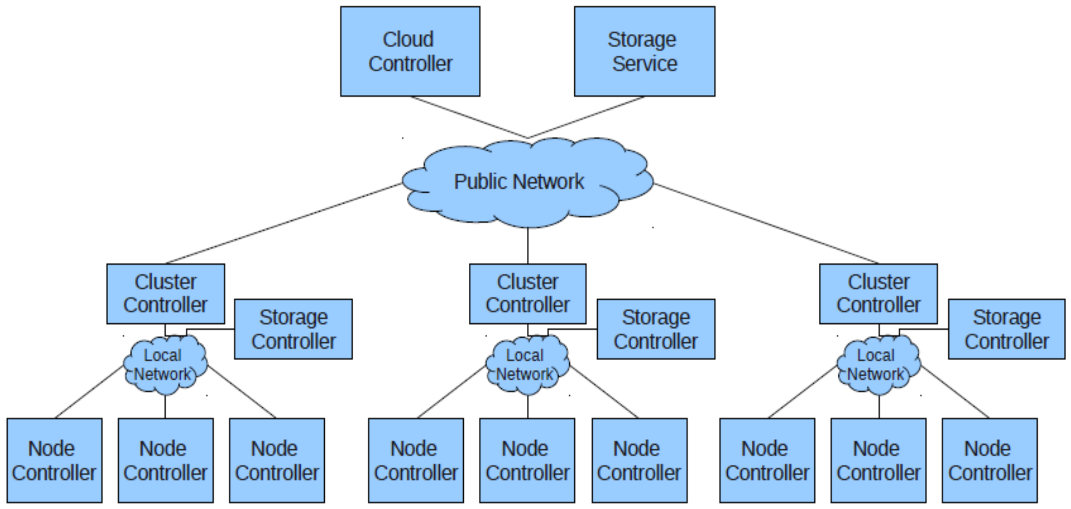


Figure 1.1: Eucalyptus Based Cloud Structure [24][25]

end users. Users then do not need to deploy local computers or other infrastructure resources. Services such as Amazon EC2 (Elastic Compute Cloud) provide such compute resources. The SaaS cloud provides an application as an Internet service to users. Users do not need to install the application locally; rather, they use it anywhere online. Examples of SaaS include Google applications and salesforce.com, which has been used for quite some time without explicitly being categorized as SaaS. The PaaS cloud is between IaaS and SaaS: it does not provide software ready to use, nor just a bare bones virtual-machine instance. It usually provides a stack of tools, server platform, and application-framework environment on top of a virtual-machine instance and exposes it as an Internet service; users can customize the provided environment and install their own software applications. This type of cloud includes Amazon S3, Bungee Connect which is designed for Cloud Application Development.

The cloud-computing approach brings more economic computing solutions. Since an enterprise does not have to put resources into building and maintaining a computer infrastructure and software applications, they can focus on their core business, such as sales, logistics, banking, insurance, etc., which provides more value to them. While companies still need some infrastructure internally, they do not need build spare capacity just for occasional demands; such demand can be managed dynamically by an external cloud. This not only saves the initial building cost, but also the operational cost. This is similar to only paying for electric usage as needed without building and running a power generation system. There is no need to buy and instal software just for short-term or occasional use; likewise, there is no need to buy each newer version. Rather, cloud computing operates on a pay-as-you-go basis and always uses the latest version, unless a prior version is required.

As the benefits from cloud computing attract more users, more IaaS implementations appeared, both from commercial sources, such as Amazon EC2, and open-source solutions, such as Eucalyptus [25][26]. Figure 1.1 shows a Eucalyptus-based IaaS cloud-implementation structure [24][25].

The Cloud Controller provides a management interface to users, collects cloud-system resource load and current-capacity information from cluster controllers, and selects the cluster controller for allocating virtual-machine instances. The Cluster Controller collects cluster-resource load and current-capacity information from node controllers, and selects the node controller for allocating virtual-machine instances. The Node Controller discovers information from the physical server on which it is running, and initiates and runs the required virtual-machine instances. The Storage Service provides file-level storage service to users to store virtual-machine images and snapshots, and allow node controllers to access these files. The Storage Controller provides remote block devices to virtual-machines instances running on the node controller.

As an infrastructure-based service provided through the Internet, the reliability and performance is critical to enterprise users. As such ecosystem, all cloud components are scaled to ensure both scalable and reliable infrastructure-service provisioning to meet demands. This type of setup usually complicates and enlarges the entire system.

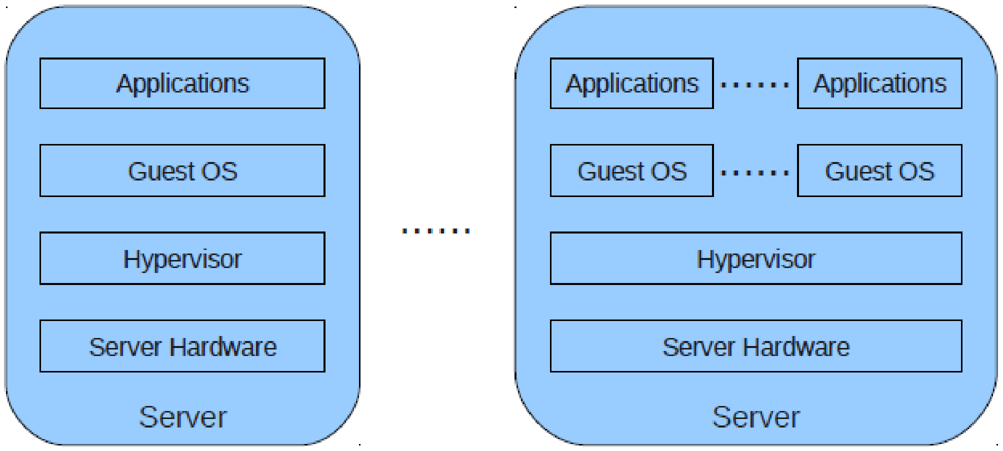


Figure 1.2: Data Center Server Structure

1.2 Data-Center Environment

Like other critical enterprise software and other Internet applications, cloud services run in computer data centers. The data center could have hundreds to thousands or more servers, depending on the needs of running applications and the expected service consumption. Some large organizations may have huge data center infrastructures – e.g., Rackspace was reported to have 63,996 servers in Nov 2010 [59], Intel had about 100,000 servers in Feb 2010 [59], Microsoft had about half a million servers [59], and Google was estimated to have over a million servers in 2007 [60].

In many traditional data centers, the servers each run an individual operating system and application for different departments or different service purposes. More modern data centers, e.g., Amazon EC2, Microsoft Azure, etc., use a virtual data-center structure, where the servers usually run some virtual-machine hypervisor (e.g., VMWare ESXi, Xen, KVM, Hyper-V, etc.). On top of that there are virtual-machine instances running similar or different guest operating systems and applications (see Figure 1.2). This setup meets more flexible usage requirements, such as dynamic computing resource provisioning, run-time online migration of the running virtual-machine instance and its application, etc.

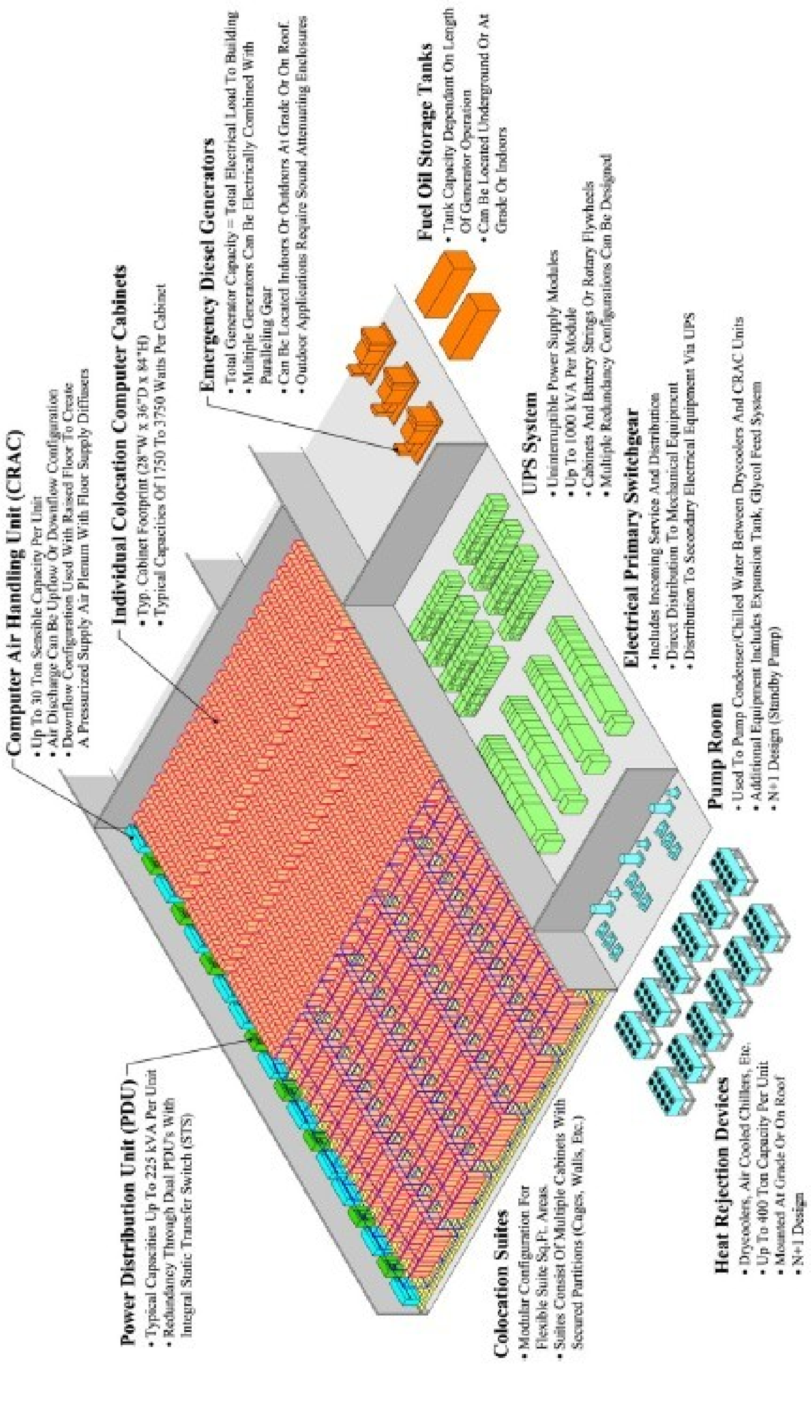

Computer Air Handling Unit (CRAC)
• Up To 30 Ton Sensible Capacity Per Unit
• Air Discharge Can Be Upflow Or Downflow Configuration
• Downflow Configuration Used With Raised Floor To Create A Pressurized Supply Air Plenum With Floor Supply Diffusers
Individual Colocation Computer Cabinets
• Typ. Cabinet Footprint (28"W x 36"D x 84"H)
• Typical Capacities Of 1750 To 3750 Watts Per Cabinet
Power Distribution Unit (PDU)
• Typical Capacities Up To 225 kVA Per Unit
• Redundancy Through Dual PDU's With Integral Static Transfer Switch (STS)
Emergency Diesel Generators
• Total Generator Capacity = Total Electrical Load To Building
• Multiple Generators Can Be Electrically Combined With Paralleling Gear
• Can Be Located Indoors Or Outdoors At Grade Or On Roof.
• Outdoor Applications Require Sound Attenuating Enclosures
Fuel Oil Storage Tanks
• Tank Capacity Dependant On Length Of Generator Operation
• Can Be Located Underground Or At Grade Or Indoors
Colocation Suites
• Modular Configuration For Flexible Suite Sq.Ft. Areas.
• Suites Consist Of Multiple Cabinets With Secured Partitions (Cages, Walls, Etc.)
UPS System
• Uninterruptible Power Supply Modules
• Up To 1000 kVA Per Module
• Cabinets And Battery Strings Or Rotary Flywheels
• Multiple Redundancy Configurations Can Be Designed
Electrical Primary Switchgear
• Includes Incoming Service And Distribution
• Direct Distribution To Mechanical Equipment
• Distribution To Secondary Electrical Equipment Via UPS
Heat Rejection Devices
• Drycoolers, Air Cooled Chillers, Etc.
• Up To 400 Ton Capacity Per Unit
• Mounted At Grade Or On Roof
• N+1 Design
Pump Room
• Used To Pump Condenser/Chilled Water Between Drycoolers And CRAC Units
• Additional Equipment Includes Expansion Tank, Glycol Feed System
• N+1 Design (Standby Pump)

Figure 1.3: Data Center Infrastructure [31]

In a modern data center, the server usually has multiple multi-core CPUs with virtualization technology, such as Intel VT and AMD-V, enabled; memory size is ranging from 16GB to hundreds of gigabytes. The servers are placed in rows of racks, and each rack usually contains around 40-48 1U-size servers. These servers are connected to the rack-level switch; rack-level switches are connected to cluster-level or data-center-level switches, which are usually modular and can support hundreds of ports and provide terabits per second (Tbps) of backbone network-switching capacity. These servers either adopt a distributed local storage or connect to a NAS (Network Attached Storage) or SAN (Storage-Area Network) through high-speed network connections, such as fibre optical channel.

While the computing components are working, they all consume power and generate heat; the accumulated heat causes many hardware problems, and eventually kills the equipments. In addition to fans running on servers and racks, there are other cooling components: CRAC (Computer Room Air Conditioning), heat-rejection systems [57], etc.

A data center either hosts the organization own services or services for their clients. Hosted services have to meet defined Service-Level Agreements (SLA). Service downtime causes business loss. Therefore high availability is required in data-center design and implementation. Not only is redundancy included in servers, clusters, and network setup, but also in cooling and power-support systems (see Figure 1.3).

1.3 Challenges and Solutions

Data-center infrastructure not only requires a large initial cost ($100 million for the Amazon data center in 2008 [53]; $500 million for a Microsoft data center in 2010 [52]), but also a high operational cost (ranging from $10 million to $14 million per year on a 125,000 sq-ft data center with 75 staff, from a 2006 survey [54]) for power consumption, service operation, and infrastructure repair.

Data center infrastructure needs considerable human resources to maintain their operations; an interview with Facebook [62] inferred a ratio of one admin staff to 130 servers. As technology develops, people need more advanced knowledge to provide proper maintenance to systems. Compared to hardware cost, well-educated people are scarce and expensive. While hardware costs continually drop, people's salary and benefits continually increase. To operate data-center services, the needed people bring significant operational cost.

To minimize cost, different research has focused on different aspects of the problem. Google [2] studied various service-request loads, the performance from cheaper commodity servers and expensive super servers, and TPC-C price/performance data; they suggest using commodity servers that could reduce the initial purchase cost and later replacement cost without impact to the provided service performance. Other than optimizing the operational process, the concept of Autonomic Computing [1][8][9][10][14] focuses on providing more automated solutions to help on relieve the human cost involved in various computer operation issues.

1.4 Automatic Recovery Problem

One of the key problems in data-center operation is to deal with server errors and failure. Bianca et al. [55] studied server failures in the Los Alamos National Laboratory. The failures they recorded only identify application interruption or server outage. Their result shows one system has almost 1200 failures per year. After the result is normalized by processor numbers in each system, it shows the average failures per year per processor is 0.3 and the worst was 0.65. They observed the failures grow proportionally with the number of processors in the system. As their study data was collected from systems built over a 9-years time frame with different technologies, they also observed that the failure rate has not changed over that period.

Translating this result to a data center with 50,000 servers each with 8 processors, it would average 120,000 processor failures each year. That means a failure would happen on average every 4.38 minutes.

Daniel [56] studied Internet data centers with a performance and availability model. In the study, he illustrated an setup of 120 servers, each with Mean Time To Failure (MTTF) of 500 minutes, and repair staff average take 20 minutes to recover the system; In order to maintain the performance standard in terms 100 out of 120 servers should be in operation, 10 staffs are needed. Translating this example to a data center with 50000 servers and each with MTTF of 50000 minutes, to maintain the same performance, more than 40 repair staffs are needed. The average IT staff now cost $44.85 per hour [51], which translates to $93267 per year by counting 40 hours per week and 52 weeks per year. So all the repair staffs will cost about $3.73 million per year to maintain this type of data center, this is just the human cost dealing with the error fixing.

The recovery is not only related with human cost, but also takes time to finish the fixing work that is represented by MTTR Mean Time To Repair, which is a downtime to the system that may even have many kinds of higher cost impacts – e.g. The lost business value ($108,000 a minute in lost brokerage operations, $43,000 a minute in lost credit card operations [58]), the penalty for SLA violation [4], or the cost induced by running the backup system or environment.

Target on the human workload and cost in recovery, Autonomic Computing, ROC [12] and container based modular data center [28] proposed different solutions. However they did not consider the situation after an error is reported, among multiple available recovery actions, which one should be choose from the overall recovery cost perspective.

The Microsoft Autopilot data center study [6] proposed a simple heuristic recovery action selection policy which simply escalate from the recovery action with minimal known MTTR to the one with highest known MTTR. It shows the consideration of the downtime cost but not specific target to the overall cost perspective. Our thesis designs a new model for automatic recovery action selection policies. The new model should be able to select a reasonable recovery action without the known MTTR of each recovery action, and it should based on a cost model.

1.5 Contribution

The work in this thesis makes the following contributions:

- we proposed a minimum total-recovery-cost model for data-center self-management. Our policy is a self-adaptive approach, bootstrapping by itself with no need for historical information. It can also add and adapt to any new recovery actions. We study and provide suggestions to shorten the initial learning curve in the model and provide preferred policy parameters based on our simulation results.

- We develop different policies from our cost-based model, and demonstrate the advantage over the Autopilot heuristic policy.

1.6 Thesis Organization

The remaining thesis is organized as following: Chapter2 provides background knowledge of the computer error and recovery systems, together with auto-recovery information. It also introduces discrete event simulation. Chapter3 describes our cost-based recovery-action selection policy and illustrates with a walk-through example. Chapter4 describes our simulator design and implementation, as well as the simulation test-data generation. Chapter5 explains the evaluation criteria and objective policy, presents the test setup, and test results. Chapter6 discusses policy improvement. Chapter7 summarizes our studies and suggests some directions for future work.

Chapter 2 Background

This Chapter provides the background information needed to understand our automatic recovery-policy research. There is substantial related work regarding computer errors and how to fix them. After clarifying the definitions of computer fault, failure and error, and related terminology, the concepts of Autonomic Computing and Recovery-Oriented Computing are introduced. These concepts provide ideas regarding management of computer failures and errors. After that, techniques used to detect and analyze errors, followed by different recovery actions are presented. Based on this background knowledge, recovery policy and related research is discussed. To help understand our experiment environment, discrete-event simulation is also described.

2.1 Fault, Error and Failure

The words “Fault”, “Error” and “Failure” are often misused. The IEEE [32] has clearly defined these, and associated terms, which we presented here for clarity.

- **System**: An entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena. These other systems are the environment of the given system. The system boundary is the common frontier between the system and its environment.

- **Service**: The service delivered by a system (in its role as a provider) is its behavior as it is perceived by its user(s); a user is another system that receives service from the provider. The part of the provider’s system boundary where service delivery takes place is the provider’s service interface. The part of the provider’s total state that is perceivable at the service interface is its external state; the remaining part is its internal state. The delivered service is a sequence of the provider’s external states.

- **Correct Service**: Correct service is delivered when the service implements the system function.

- **Service Failure**: A service failure, often abbreviated here to failure, is an event that occurs when the delivered service deviates from correct service.

- **Service Outage**: The period of delivery of incorrect service is a service outage.

- **Error**: A service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an error. The definition of an error is that part of the total state of the system that may lead to the system's subsequent service failure. It is important to note that many errors do not reach the system’s external state and cause a failure.

- **Fault**: The adjudged or hypothesized cause of an error is called a fault.

- **Partial Failure**: When the functional specification of a system includes a set of several functions, the failure of one or more of the services implementing the functions may leave the system in a degraded mode that still offers a subset of needed services to the user. The specification may identify several such modes, e.g., slow service, limited service, emergency service, etc. Here, we say that the system has suffered a partial failure of its functionality or performance.

From these definitions, we know that error is defined in the entire system scope and may not cause a service failure which shows in the external state. In our research we assume an error detector which reports any errors in the system that may cause the service to deviate from the defined specification, causing either full or partial service outage. In this way, we simplified our discussion and the implementation of simulation just to two parties – error or failure is what has been detected and cause service away from SLA, fault is what behind the error as the root cause.

2.2 Autonomic Computing

The Autonomic Computing concept was created by IBM researchers [8] to address management costs in complex computer environments. While computer systems provide more services, they have also become more complex and bulky. As systems continuously expand, the size and complexity is approaching or exceeding human management capability. System operation and management costs are significant to businesses. Autonomic computing believes the solution is to have the system self manage, with operators specifying policy.

2.2.1 Autonomic Function Aspects

Autonomic computing defines system self-management as addressing four functional aspects:

- Self-Configuring
- Self-Healing
- Self-Optimizing
- Self-Protecting

Self-Configuring

A computer-system environment in a data center is usually composed of different hardware (switch, router, firewall, server, load balancer, storage array, etc.) and software (operating system, database, web server, application server, framework utility, application, etc.). The integration and configuration of these components is complex and error-prone.

A self-configuring system is one in which this work is done automatically by the system. New components register to the system, which can also discover, configure, and adapt to the existing components, meanwhile existing components can discover, configure, and adapt to new components. They automatically melt together and form new system capabilities. This adaptive process is continued for any environment changes at run time.

Self-Healing

The hardware and software components in a data center have errors. These errors have not only a cost to fix, but also the opportunity cost caused by system downtime.

The idea of self-healing is to let system overcome any errors encountered by itself, without significant downtime. The system should be able to detect any errors, analyze the cause if necessary, and recover without noticeable service interruption. Recovery-oriented computing provides some ideas and techniques in this area. Our research also address solution for automatic recovery.

Self-Optimizing

As many parameters exist in both hardware and software components, adjusting these parameters to achieve optimal performance introduces a lot of issues. The optimization of some parameters in a system may have a negative impact on the performance of other systems. Experts familiar with subsystem optimization are scarce and expensive. To optimize and balance the overall system performance is more complex still.

Self-optimizing requires the system to adjust itself for the best performance. The system should be able to identify optimization opportunities under a changing environment, analyze the impact, and optimize the system with minimal overhead.

Self-Protecting

Even though firewalls and other security appliances are installed, malicious attacks still happen. The consequence of a successful attack is loss to the organization.

Self-protecting has the system anticipate, detect, and defend itself against wide attacks and cascade failures. It should also correct the whole system from any attacks that can not be recovered by self-healing. This capability further reduces operational cost.

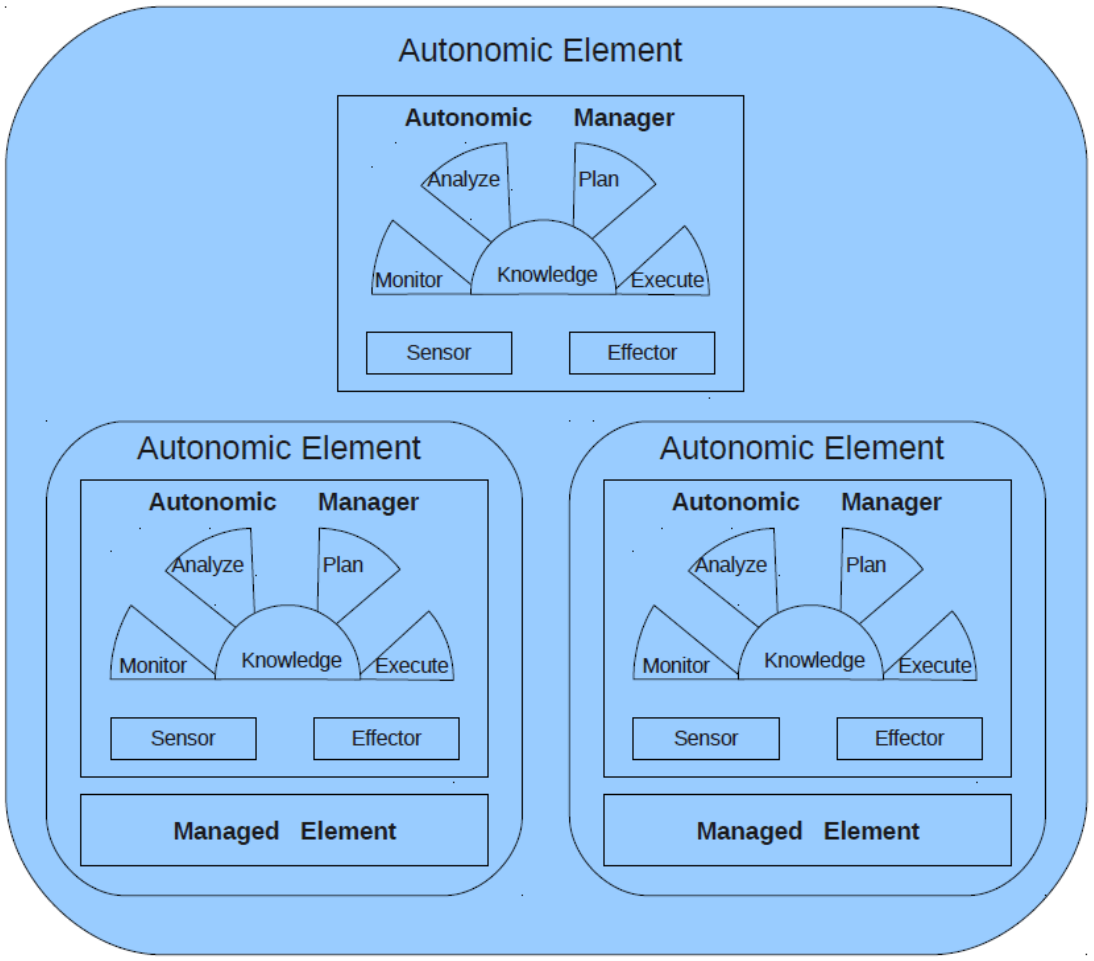


Figure 2.1: Autonomic Elements

2.2.2 Autonomic-System Architecture

The proposed autonomic-computing architecture [8][11] is a recursive structure of components (see Figure 2.1). Each component is called an Autonomic Element. Each element should implement the four autonomic-computing functions, making each element a small self-managed system. The Autonomic Element is the building block to form a larger self-managed system. The formed structure could be a tree or an arbitrary graph.

An Autonomic Element contains an Autonomic Manager and one or more Managed

Elements. Inside the Autonomic Manager there are sensors collecting data from the Managed Elements; a monitor watches the data collected by the sensors and filters the data for analysis; from the analyzed result an execution strategy is planned and executed; knowledge required for this is shared as necessary; finally the effectors execute the desired actions in the Managed Elements.

The autonomic-computing concept defines the desired system capability to address many issues in today's complex computing environment. Our cost-based automatic-recovery policy provides a practical solution for the self-healing aspect, as well as addressing some aspects of self-configuring and self-optimizing (e.g., it can adapt to new recovery actions, and it continuously optimizing for the selection of the best recovery action). Our policy implementation is similar to the autonomic-component structure.

2.3 Recovery-Oriented Computing

Another approach for dealing with computer failures, errors, and faults is Recovery-Oriented Computing (ROC), which focuses on fast recovery from failures and offers high availability for Internet services.

2.3.1 Motivation

In the past several decades, the price to purchase computers has dropped significantly. In contrast, the cost of human intervention is higher and higher. Meanwhile, the capabilities of hardware and software result in more users of these systems, and more system administration work is involved. As a result, more and more failures of systems are caused by human, such as system operators, or system administrators. The surveys on public switched telephone network and three internet sites show that the operators cause 59% and 51% failures in those systems respectively [12][38]. Not surprisingly, the total cost of ownership (TCO) remains high, and a large portion of the TCO is from human operator error. ROC is hence motivated to achieve higher availability and lower TCO.

2.3.2 Principal and Hypotheses

ROC states the following principal and hypotheses [33][12]:

- Failure modes are not predictable. Failures are caused by imperfect and incompletely modelled systems; thus they can not be foreseen. In addition, David Patterson et al. [12] stated ROC's guiding principle – the failures, faults, errors are facts to be coped with, not problems to be solved.
- Recovery performance is more important than computing performance
- TCO is more important than software and hardware purchase cost

By concentrating on Mean Time to Repair (MTTR) rather than Mean Time to Failure (MTTF), ROC offers higher recovery performance and system availability. Faster recovery time impacts Total Cost of Ownership (TCO) as well, so that it improves system dependability.

2.3.3 ROC Techniques

Online Verification of Recovery in both development and production environments. Unlike functions that are well tested before put into production, recovery is hardly verified. This causes the case in which recovery is not working in production. However, it is difficult to test real faults in these environments. FIG (Fault Injection in glibc) [12][34] provides a lightweight, extensible tool for simulating, injecting, and logging errors in a running system. It not only can be used in development environments to make the system more robust, but also can be used in production environments to expose environment-related errors that are unable to be detected in the testing phase. The verified environment is then more likely to correctly recover from errors to achieve higher dependability.

Facilitate diagnostic support. Complex computing environments make it challenging to detect system problems and identify their root causes quickly. Traditional tools rely on static dependency models which can not keep up with the changing environments. Pinpoint [12][36], a diagnostic tool implemented in J2EE, enables error detection and analysis without any knowledge of the systems being monitored or requests. It records the components that are used to response each individual client request, detects internal faulty components and external end-to-end failures, and distinguishes the faulty components based on the collected data. This greatly helps administrators to rapidly recover from system failure, resulting in lower TCO.

Partitioning for rapid recovery. To recover a large-scale system in one shot can be expensive, resulting in service disruption and long downtime; critical systems can not tolerate this disruption. Partitioning the system to modular and lower coupling subsystems can contain the error locally, causing less impact to the overall running system. Microreboot [12][13][35][37] adapts this technique to partially restart the affected components of a system without interrupting the whole system, and masks failures from end users. This practical ROC technique achieves low-cost recovery.

Undo enabled system with safety margin. Human error may be caused by performing incorrect actions or performing actions correctly but not intentionally. This type of human error is usually hard to correct in ordinary systems. System with Undo capability enabled [12][35][40][41][42] support retroactive repair and recovery from human errors, thus improving system dependability.

Undo is based on three R's of rewind, repair, and replay. In the rewind step, all system state is reverted to the time before the error happened. In the repair step, the operator takes actions to correct the mistakes or prevent the errors. Finally, in the replay step, all the user interactions are executed again to make sure the information between the rewind point and the present would not be lost.

Defence in depth. Cascading failure may easily crash normal protection mechanisms; with more levels of protection, a system may survive from even worse failure, that makes the system more dependable. For example, multiple levels of firewalls and virtual machines on top of real machines make systems safer and more robust [12].

Redundancy and fail fast. In the physical world, the production of physical objects involves non-negligible cost and time. However, in software systems, multiple component instances can be created with no substantial cost. Redundant component instances can avoid a single point of failure. Moreover, when a component fails, kill it and start a new one is frequently much faster and cheaper than diagnosing the root cause and fixing it. [12]

2.4 Error Detection and Analysis

Being aware of an error in the system is the first step in automatic recovery. There is much research focused on error detection, and many approaches for figuring out the root cause behind the error. Some research also considers the overhead of doing error detection and analysis, and seeks solutions to minimize it.

2.4.1 Supercomputer System Log Study

Oliner et al. [19] studied system logs from five supercomputers in order to explore issues that should be considered in automatic error detection. The log study itself was done offline. After collecting the logs, alerts were identified by using regular expressions. Due to many duplicate alerts, a temporal-based filtering was performed. Finally, the modelling of failure timing was conducted.

From this study, some recommendations were provided for automatic-detection-related research. The big issue for accurate error detection is missing operational context: this information will help understanding the expected system behaviour. There are more difficulties in finding the root cause: the logging system itself may be corrupt; many duplicated or similar alerts could be triggered by one error; different categories of error have different signatures; the system continuously evolves; filtering under these issues is also difficult. To analyze the root cause accurately needs further research.

2.4.2 Early Warning Principles

Dorron et al. [18] studied alarms logged in telecom voice-mail systems. Due to the different customer-environment configurations, in stead of providing a rigid algorithm, they provided three principles to achieve early warning of system failures.

The first principle is to simply count all alarms in the system. By averaging the counting number in a period, a stabilized number can represent the normal operation of the system; an increased number indicates some developing problem, whereas a decreased number shows some fixes were done. This principle can not point out the potential faulty sub-system, but gives early warning for the over all system which is still helpful for bringing attention and applying early actions.

The second principle is to count alarms for each sub-system individually, mix and rank them into a Pareto diagram. The stabilized Pareto ranking presents normalcy; ranking changes and an increased alarm counter of a sub-system indicates a potential failure of that sub-system. Also, a similar approach can be used to count alarms from different alarm IDs inside each sub-system; ranking changes for different alarm IDs indicates a problem with that sub-system. This principle provides more accurate warning to the sub-system level.

The third principle is to count clusters of some alarms. Application and operation experts can usually determine some appropriate grouping of alarms; the alarm count from these selected groups can provide a much earlier warning for the related sub-system.

These principles were applied to customer systems, and successfully demonstrated early warning for system failures. While this study is interesting, it is not a fully automatic solution and can not be directly used in a generic system.

2.4.3 Adaptive Monitoring With Statistical Models

A series of studies [15] [16] [20] were conducted by Munawar and Ward. They utilized linear regression statistical models to detect errors and analyze faults in a generic application environment (J2EE). They also utilized an adaptive-monitoring approach to minimize the monitoring and analysis cost.

This monitoring involves three phases. The first phase is model building. It is assumed there are no faults in this phase; the target system is run and metrics are collected. Based on those metrics linear correlation models are learned. A subset of the most representative metrics and models are selected based on whether they are associated with a majority of the system components. The system is then ready for minimal monitoring. This monitoring phase is based on a small subset of metrics and models, so the cost is minimal. In this phase, if deviation from the model is observed, an anomaly is deemed to be happening in the system. Then system gets into detailed monitoring, so as to determine if the anomaly is a glitch or significant. In this phase, system collects and studies all learned metrics and models related with those deviation metrics and outlier models. From the intensive data collections, the components associated with relative more outlier models were selected to a suspected faulty component list with rankings. This phase incurs relative higher cost to the system, however only happens in a short period when system is suffering error.

With a minimum impact to the system, the results show that errors generated by 28 of 29 injected faults are detected, and the faulty component is shortlisted 65% of the time. While this is almost perfect for automatic error detection, fault analysis only produces a suspected-faulty-component list, not the exact component and not the root cause behind that fault.

2.4.4 Integration of Monitoring Data

Munawar and Ward also proposed an architecture [17] to integrate monitoring data from all sub-systems. These heterogeneous systems have different monitoring data types and formats. The proposed solution adds parsers and other conversion engines to different sub-systems and data sources, and converts the source data to the Common Base Event format for problem analysis. Continuous monitoring data (e.g., CPU utilization) is not suitable for the discrete CBE data format; it is also not feasible to be continuously collected due to the high collection cost. By utilizing their previous adaptive-monitoring work, they create a continuous data monitoring engine. The engine keeps monitoring the system for those numeric data at a minimum level; when a problem is detected, it is then reported using a behavioural model which is CBE compatible; detailed monitoring can be engaged for further analysis. The engine makes the conversion of continuous monitoring data feasible. This architecture integrates all types of data from all sources, providing a more comprehensive picture to the event analyzer for better system analysis.

The previously described ROC PinPoint system is also an error-detection solution. The results from this research are encouraging. We noticed that most research did well on error detection; however they are not reliable on determining the root cause. No current solution is able to point out the exact fault behind an error; they usually can only give a range of faults.

Our research is not focusing on the error-detection area; instead, we use the error-detection result. We rely on a good error detector which not only notices every error requiring recovery, but also provides feedback on whether or not the previous recovery succeeded by noticing if anew error happens shortly.

2.5 Recovery Actions

After an error is detected, we need a recovery action to solve the problem. We describe some automatic recovery actions in this section.

2.5.1 Reboot and Micro-Reboot

One of the automatic recovery actions is reboot. Desktop computer systems were not very stable in the past, they may hang in a couple hours. Pressing the reset button is a quick and effective way to make the system work again. In the automatic approach, we just need to send a reset signal if the system has completely hung, or send a reboot command if the system is still responding at a certain level.

During the restart process, if the system needs load too many libraries and drivers, set up the environment, and run batch jobs for loading many applications, the reboot may still take a long time which means high cost. Micro-reboot [12] [13] realizes the ROC partitioning technique by rebooting a part of the system not the whole system. For example, it may restart certain processes or components. However, this approach may require some fault information to be provided regarding where the fault is: in which process or component.

2.5.2 Re-Image

Another automatic recovery action is to re-image the system. After the system has run for a long period it may show problems caused by corrupt files or minor disk errors. The re-image approach [6] is to format the disk, reinstall the Operating System, and then reinstall applications. The freshly reinstalled system usually makes those problems go away. Virtual-machine technology makes re-imaging even simpler. Since the whole virtual-machine instance, including OS and applications, is just a virtual disk, either an ordinary file or a disk file; replacing that file can achieve the same re-imaging result as the traditional way.

2.5.3 System Rejuvenation

For long running systems, the phenomenon of software aging usually leads to system degradation, possibly crash the entire system. The causes of this degradation may be software bugs or unexpected errors, memory leaks, data corruption, unreleased file locks, etc. Unlike Reboot, which takes action after a system failure, Software Rejuvenation [44] prevents system degradation and failures in the future by proactively terminating the system, cleaning its internal states, and then restarting the system. This can be done at a scheduled time, potentially resulting in lower cost.

The methods to determine and optimize the rejuvenation schedule vary. For example, some studies [44][45] used a time-based rejuvenation model, Bobbio et al. [48] optimized the rejuvenation policies by using fine-grained software degradation models, Garg et al. [47] considered the time and workload as the factors to model the rejuvenation policies, Vaidyanathan et al. [46] used Stochastic Reward Nets (SRNs) for the rejuvenation model under a cluster environment.

2.5.4 Roll Back

The rollback recovery action takes a checkpoint (also called a log) of the system state at various times when the system is assumed to be healthy. When errors are detected, a recent checkpoint can be used to restore the system to a healthy state.

A typical utilization is the database check pointing mechanism [49][50]. It takes a snapshot of a healthy database and log the data at a given time interval; when the database is crashed, it will restore and restart the database using those checkpoints, which can limit the recovery time and restore the most recent user data, hence avoiding significant business loss.

2.5.5 Other Recovery Actions

Other than the actions we described above, there are still other recovery actions such as: release disk space by deleting or moving files in directories that can fix system failure due to a disk-space shortage; this may be a lower cost recovery action compared to reboot; upgrade operating systems and install other software patches to fix some software bugs.

2.6 Recovery Policy

After errors has been detected and reported, the question to be addressed by the recovery policy is what action should be used? If we know what is the fault behind the error, and what actions can fix which faults, what is the cheaper action among them, it would be easy to select the action. Without knowing exactly the fault, what is the best action among several candidates? So far just a couple research papers have focused on this area.

The recovery action done by a human expert usually has the most expensive recovery cost, although it is assumed they can always fix any problems. We want to save the cost by reducing request to a human. By way of analogy, in our life, if any glitches happen in our car we go and see the mechanic; that would cost a significant amount of money for an old car or we may not even be able to afford it. The usual solution is to try a couple of easy and cheap techniques we know; only the remaining unsolvable problems will go to the mechanic. This way we can avoid a lot of unnecessary visits to the mechanic and hence save money and time. A similar idea can be applied to computer system recovery: let the computer system apply several low-cost automatic recovery actions to the error, and observe the result; if the

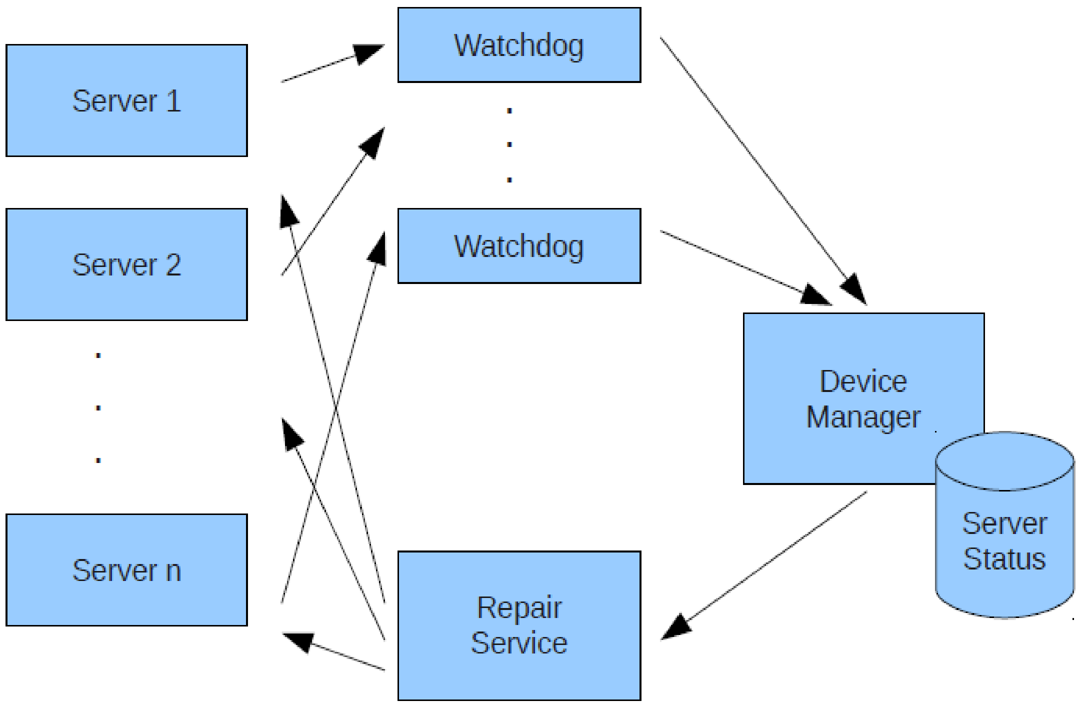


Figure 2.2: Data Center Automatic Recovery

problem still exists, the operator will finally be notified for further action. The Autopilot solution [6] adopts this concept.

2.6.1 Autopilot Recovery System

The Microsoft Autopilot system provides a complete automated data center management solution, which includes automatic software deployment and provisioning management, as well as automatic recovery management. As our study is focusing on automatic recovery policy, we only show that in Figure 2.2.

In the system, each server is monitored by one or more watchdog. Some of the watchdogs monitor different functions on the server, such as memory, disk, etc. Some of the watchdogs monitor different applications on the server. A watchdog can report “error”, “OK”

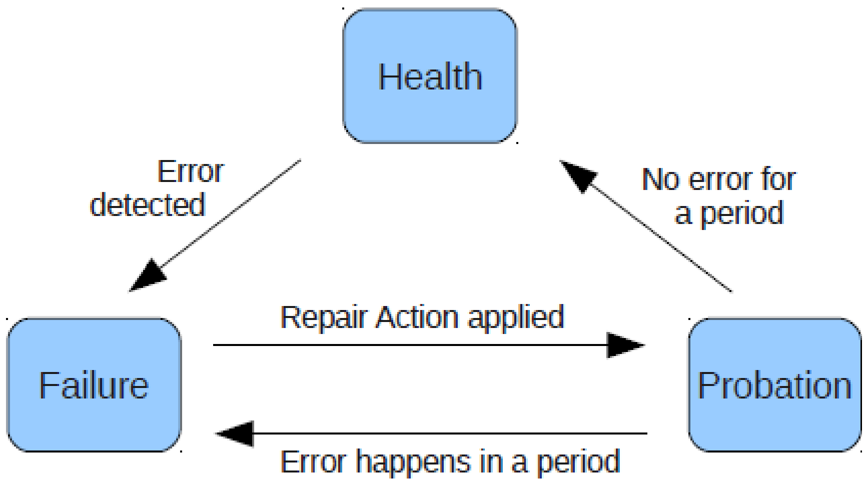


Figure 2.3: Server State Change

or “warning” to the central Device Manger. The Device Manger deems a server has an error as long as there is one watchdog associated with that server which reports an error.

The Device Manager maintains the status of all servers (Figure 2.3). When an error is detected on a server, the server is marked as “Failed”. After a recovery action is selected and performed on the server, the sever state is moved to Probation. If no error happens in probation for a defined period, the server is deemed healthy. If an error happens while the server is in the probation state, the server state is moved back to Failure.

The Device Manager selects a recovery action from the list: Do-Nothing, Reboot, Re-Image, Replace/Go Operator. The decision is based on the error type and recent server history. If the error is a fatal error, like disk or other hardware error, the recovery action Replace/Go Operator will be selected. If the error is non-fatal, and the server has had no error for a long period, the Do-Nothing action will be selected based on the assumption that the error is temporary. However, the server will be placed in probation. If there is another error on the server in a short period (while it is in probation) the Device Manager will escalate the selected recovery action from Do-Nothing to Reboot, to Re-Image, and eventually to Replace/Go Operator. These selected recovery actions will be performed by Repair Service on the problematic server.

2.6.2 Autopilot variations and limitations

Moises et al. [3] added the recovery actions (non-destructive re-imaging, destructive re-imaging, software upgrade, etc.) but still used the Autopilot recovery action selection policy, refining it with a survival analysis and statistical machine learning. Guy et al. [5] took a similar approach but their refinement used a Partially Observable Markov Decision Process. These techniques provided a refinement on a base policy but did not provide any new policy model. However the base policy has the following problems:

1. In the recovery action escalation ladder, the higher recovery action can resolve all issues the lower action can resolve. This may not always be true, particularly when more recovery actions are available; the potential effects of various actions may overlap or may be completely unrelated.
2. It assumes that the higher recovery action will cost more than the lower one. This may not be true, when more actions are available.
3. These assumptions imply the recovery cost and effect are clearly known in advance. This restricts the initial adoption and future dynamic expansion of the policy.
4. Another implication from the above assumptions shows the cost and recovery effect has linear relationship. This may not always be true, one recovery action may cost the same or less but can resolve more problems comparing to another action. For example, in virtual machine environment, the re-image could be as fast as reboot but with more recovery effects.
5. Moreover, the initial choice is always fixed with Do-Nothing, this may be a waste of time in terms of cost.

Autopilot solution does consider to select the recovery action based on the lowest cost, however the selection is not adjusted on the real recovery action cost and without considering the real recovery effects. We will address these problems and provide a new base policy model.

2.7 System Simulation

To study different policies, the best environment is a real computer data center with thousands of servers and running different policies for long periods to collect data for comparison. Due to the quantity of the servers involved and the critical services they support, this kind of environment is not feasible for our experimental purpose. A simulation approach is therefore preferred.

While there are several cloud simulation systems, such as the Open Cirrus Testbed [30], CloudSim Toolkit [21] [27], SPECI [22], and others [29], they do not focus on, or even consider, automated recovery in their designs. We therefore decided to design a discrete-event simulator for our research, hoping it could also benefit other similar research.

Chapter 3 Total Cost Based Policy Design

Following the Autopilot approach, without knowing the exact fault or any fault information at all, we wish to select the best recovery action among a list of recovery actions. While still using the Autopilot three-state recovery mechanism, is there a better policy than the Autopilot simple heuristic policy? Can we know the recovery-action cost more accurately than just guessing? Can a policy be made based on a generic cost model and reduce the overall recovery cost? Our automatic recovery-action selection policy research is focused on these questions. First we create a cost-based model for recovery-action selection. Then we define policies to use that model. Finally, we walk through a concrete example to show how it works.

3.1 Problem Modelling

Our approach to the problem is to model the effects of recovery actions on errors. We do so by modelling the probability that any given recovery action will fix a fault. We then study the problem of recovery action cost.

3.1.1 Modelling Recovery Probabilities

Suppose a system has faults f_1, f_2, ... f_n and recovery actions R_1, R_2, ... R_r. Recovery action R_i has some probability P_i_j of fixing fault f_j. There are different ways to model this. One way is to say that R_i either succeeds or fails. Thus, the probability that R_i fixes f_j is one of {0,1}, and nothing else. In such a model, there is no point in trying recovery action R_i twice. It either succeeds or it cannot. Alternately, a recovery action may have some non-perfect, but non-zero probability of success, in which case it might be reasonable to retry the recovery action.

Examples of the first case are any system in which faults are deterministic. For example, a car will not move. The underlying fault is that the car is out of gas. Filling up the tires with air has zero probability of solving the problem; adding gas to the car has 100% probability of solving the problem. Filling up the tires with air twice will not make the situation any better.

Examples of the latter case are any system in which faults are non-deterministic. For example, a loose electrical connection in a plug; pulling out the plug and putting it back in may succeed with some probability that is less than 1 but more than zero. Retrying this procedure a few times may be a reasonable course of action.

Similarly, there are examples in software for deterministic faults. For example, a service fails to run properly. The underlying fault is that there is a file corrupted. Rebooting the system has no chance to solve the issue at all; re-imaging the system can always solve this issue.

There are non-deterministic fault examples in software as well. For example, a web application will display newly added customer data in the last 5 minutes in one page; some times the page can not be displayed. The underlying fault is there is not enough processing memory for producing all the data in one page. Doing nothing some times works, because there is less customer data added in the previous 5 minutes. Some times it does not work because more customer data is added suddenly. A similar result happens with reboot. However, rollback to a previous version may make the system work better, because there is more memory allocated in the previous version. Although it works better, it still may fail under extreme customer data input volume. Upgrading to a newer version with a paging function built-in can completely resolve the issue. So the Do-Nothing, Reboot, and Rollback recovery actions have a certain probability to make the problem disappear for a period of time, but will not always succeed.

In the situation where the fault behind an error is unknown, suppose a system has recovery actions R_1, R_2, ... R_r and recovery action R_i has some probability P_{Ri} of fixing errors in the system. This probability is the overall recovery-action success probability in the system. From a deterministic fault perspective, this system-wide success probability actually represents the fault distribution; the success probability is the percentage of the encountered errors with faults which the recovery action can always fix. Conversely,

$1-P_{Ri}$ is the percentage of the encountered errors with faults which the recovery action can never fix. From a non-deterministic fault perspective, the P_{Ri} system-wide success probability is the combination of fault distribution and the recovery action effectiveness. Without more fault specific information, it is not possible to know what fraction of the probability is due to the fault distribution and what is due to non-determinism.

3.1.2 Recovery-Action Cost

In order to create a cost-based model, first we want to look at what is the cost of the fault and what types of cost are involved in recovery. As mentioned in the introduction, the service outage in terms of downtime may cause many types of cost, such as business loss, human cost, etc; however, they are all a consequence of the service outage. They may have a different relationship with the service downtime in different organizations. Due to the dependency to the individual organization and the derivation from service downtime, we simplify the cost as service downtime, and assume this includes the recovery cost as well as the cost due to the downtime.

When recovery action Ri is applied to an error, it may succeed or fail. The different results will introduce different costs: Cs_{Ri} cost of recovery action Ri succeeded and Cf_{Ri} cost of recovery action Ri failed.

For the average cost of using recovery action Ri and it succeeds, the recovery action success unit cost UCs_{Ri} :

$$UCs_{Ri}=\frac{\sum_{m=0}^{As_{Ri}} Cs_{Ri}(m)}{As_{Ri}}$$

Where As_{Ri} is how many attempts the recovery action Ri succeeds.

For the average cost of using recovery action Ri and it fails, the recovery action fail unit cost UCf_{Ri} :

$$UCf_{Ri} = \frac{\sum_{n=0}^{Af_{Ri}} Cf_{Ri}(n)}{Af_{Ri}}$$

Where Af_{Ri} is how many attempts the recovery action Ri fails.

Therefore, the average cost of a recovery action is:

$$UC_{Ri} = P_{Ri} * UCs_{Ri} + (1 - P_{Ri}) * UCf_{Ri} \tag{3.1}$$

3.2 Generic Cost-Based Model

As we know the average cost of each recovery action, and the probability they may succeed in the system, we are able to choose a recovery action with the least cost. However, the recovery action unit cost does not represent the total cost to fix the problem. The recovery action may fail; if it fails, we may try it again or introduce another recovery action. What then is the total cost to fix a problem by using various recovery actions?

3.2.1 Estimated Total Cost of Recovery (2 actions case)

We start from a simple case with recovery action R1 and recovery action R2 only, and their unit cost and success probability are known. The total cost to recover a problem by selecting recovery action R1 depends on the cost if it succeeds and the cost if it fails. We first consider deterministic faults, so if action R1 fails, it is not worth retrying it. Therefore, the estimated total cost of selecting action R1 first is:

$$\begin{aligned} ETC_{R1} &= UCs_{R1} * P_{R1} + (UCf_{R1} + ETC_{R2}) * (1 - P_{R1}) \\ &= UCs_{R1} * P_{R1} + UCf_{R1} * (1 - P_{R1}) + ETC_{R2} * (1 - P_{R1}) \end{aligned} \tag{3.2}$$

When action R1 fails UCf_{R1} is incurred, and further recovery actions need to be used to finish the recovery. Under the failed case, as R1 has zero probability of success, the other recovery action R2 needs to be involved, and the similar ETC_{R2} total recovery cost of recovery action R2 is incurred. According to Equation 3.1, the final ETC_{R1} estimated total cost of recovery action R1 can be simply expressed as Equation 3.3.

$$ETC_{R1}=UC_{R1}+ETC_{R2}*(1-P_{R1}) \tag{3.3}$$

However, since R1 has been tried and failed, only action R2 is left. In this situation, recovery action R2 has:

$$ETC_{R2}=UC_{R2}$$

Hence, the estimated total recovery cost of selecting action R1 first becomes:

$$ETC_{R1}=UC_{R1}+UC_{R2}*(1-P_{R1}) \tag{3.4}$$

Similarly, the estimated total recovery cost of selecting action R2 first is:

$$ETC_{R2}=UC_{R2}+UC_{R1}*(1-P_{R2}) \tag{3.5}$$

The decision can be made to select action R1 first if

$$\begin{gathered}ETC_{R1}<ETC_{R2}\\UC_{R1}+UC_{R2}*(1-P_{R1})<UC_{R2}+UC_{R1}*(1-P_{R2})\\UC_{R1}+UC_{R2}-UC_{R2}*P_{R1}<UC_{R2}+UC_{R1}-UC_{R1}*P_{R2}\\-UC_{R2}*P_{R1}<-UC_{R1}*P_{R2}\\UC_{R2}*P_{R1}>UC_{R1}*P_{R2}\end{gathered} \tag{3.6}$$

The simple cases of this comparison are: if both action R1 and R2 have the same success probability, then the action with lower average unit cost will be selected first. Conversely, if both actions have the same average unit cost, the one with higher success probability will be selected first.

No matter which recovery action is selected first, after both actions are tried and failed, a same failed chance is still left:

$$(1-P_{R1})*(1-P_{R2})$$

In this case, the order of selection is irrelevant to the actual cost, since the cost will be $UCf_{R1}+UCf_{R2}$. However, in general some recovery actions should succeed, in which case their costs and probabilities of success do affect the preferred order of action selection.

In addition, the success probability when both recovery actions are applied is:

$$1-(1-P_{R1})*(1-P_{R2})$$

3.2.2 Estimated Total Cost of Recovery (3 actions case)

We now consider a more complex case. Where there are 3 recovery actions, so there are more choices if R1 has failed. Instead of the only estimated total recovery cost of selecting action R1 first in previous example, there are various estimated total recovery cost of selecting action R1 first according to rest selection sequence in this case.

R1 -> R2 -> R3

Or

R1 -> R3 -> R2

When R1 is first selected and has failed, there is a choice to further try R2 or R3 secondly:

$$ETC_{R1} = UC_{R1} + ETC_{R2} * (1 - P_{R1})$$

Or

$$ETC_{R1} = UC_{R1} + ETC_{R3} * (1 - P_{R1})$$

Clearly, to get the lowest ETC_{R1}, the minimum between ETC_{R2} and ETC_{R3} under R1 failed situation will be chosen. Assume R2 is chosen as second action and also fails, then R3 could be last tried.

$$ETC_{R2} = UC_{R2} + ETC_{R3} * (1 - P_{R2})$$

At this point, both R1 and R2 has been tried and failed, so no more action if R3 will also fail, then the recovery action R3 has:

$$ETC_{R3} = UC_{R3}$$

Hence, After recovery action R1 has failed, the estimated total recovery cost of selecting R2 as second action becomes:

$$ETC_{R2} = UC_{R2} + UC_{R3} * (1 - P_{R2})$$

Similarly, After recovery action R1 has failed, the estimated total recovery cost of selecting R3 as second action becomes:

$$ETC_{R3} = UC_{R3} + UC_{R2} * (1 - P_{R3})$$

The decision can be made to select R2 as second action if:

$$ETC_{R2} < ETC_{R3}$$
$$UC_{R3} * P_{R2} > UC_{R2} * P_{R3}$$

As the second recovery action with minimum recovery cost selected, the ETC_{R1}

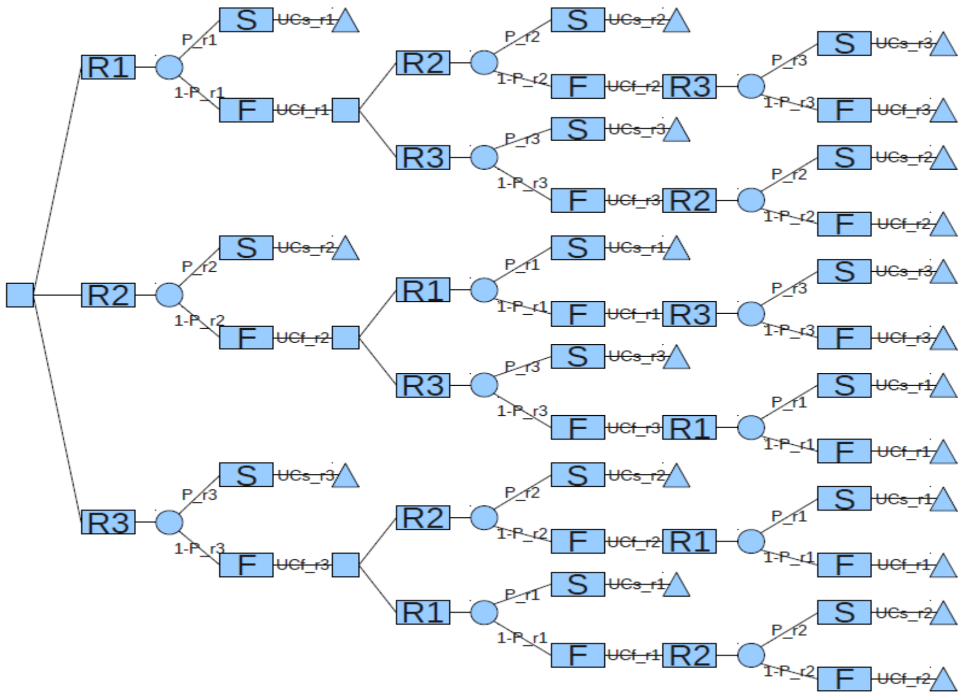


Figure 3.1 Recovery Action Selection

estimated total recovery cost of selecting action R1 first will have the lowest cost among its varieties. The same procedures also apply to deal with the various estimated total recovery costs of selecting action R2 and R3 first. After their second recovery action selected respectively, the best estimated total recovery costs of selecting R1, R2, and R3 first are also generated. By comparing these best values of ETC_{R1}, ETC_{R2}, and ETC_{R3}, the recovery action with the lowest cost will be selected to use first. These choices are illustrated in Figure 3.1. To determine the minimum estimated cost, we must search this tree.

Again, no matter which recovery action is selected first, and second, after all three actions are tried and failed, a same failed chance is still left:

$$(1-P_{R1})*(1-P_{R2})*(1-P_{R3})$$

And, the success probability when all three recovery actions are applied is:

$$1-(1-P_{R1})*(1-P_{R2})*(1-P_{R3})$$

3.2.3 Minimum Estimated Total Cost of Recovery

We now extend the recovery actions to an arbitrary number. The estimated total cost of recovery will vary according to recovery action selection sequence (see Figure 3.1). In two actions case, there is one level of decision to make. In three actions case, there are two levels of decision to make. When more recovery actions are available, it will have more levels of decision to make. However, in each level of decision, it is the same approach to select the minimal estimated total cost among the candidate recovery actions under the same condition that the recovery actions thus far have failed. Therefore, we solve this using a recursive search on the tree, the minimum estimated total recovery cost based action selection process can be summarized as:

$$\begin{aligned} \min_{Rx\in A} ETC_{Rx} &= UCs_{Rx}*P_{Rx}+(UCf_{Rx}+\min_{Ry\in A,\, Ry\neq Rx} ETC_{Ry})*(1-P_{Rx}) \\ &= UC_{Rx}+\min_{Ry\in A,\, Ry\neq Rx} ETC_{Ry}*(1-P_{Rx}) \end{aligned} \quad (3.7)$$

This formula hence becomes our generic cost-based model for recovery-action selection. Among a set of recovery actions, the one with minimum estimated total cost is the choice for recovery with lowest cost consequence.

This formula is derived from a deterministic perspective. The action recovery success probability is actually the fault distribution of encountered errors, and either some action will succeed or none will succeed:

$$P_{\text{none}}=\prod_i (1-P_{Ri})$$

$$C_{\text{none}}=\sum_i UCf_{Ri}$$

If one succeeds, it depends on the recovery action selection sequence as to what the total estimated recovery cost is. The selection sequence is based on probability of success (i.e., fault distribution) and recovery action cost.

Along with the recovery action unit cost and recovery action success probability modelling, to simplify this generic cost-based model, the following assumptions were made:

- The model and policy only deal with non-fatal errors.

This assumption is the same as in the Autopilot heuristic model. The different selections in their model are only for non-fatal errors; as for fatal hardware errors, there is no real fix choice other than to go to the operator.

- The model is based on deterministic types of fault and recovery actions.

The second assumption is to simplify the recursive selection – under a recovery action failed case, that recovery action is not worth trying again, because we are modelling it as either succeeding at recover the fault with probability 100% or failing with 0% probability. This makes the recovery action itself not be the candidate recovery action when it fails, the further selection under its failed situation will have less recovery actions to choose; and until a level there is no more choice for candidate recovery actions, the recursive selection then ends.

If it is based on non-deterministic model, there are two complicated issues here: the first issue is that the recursive selection becomes unlimited and will not end, because there is always a chance to retry the same recovery action even if it has failed; the second issue is that the accurate success probability of the recovery action under its failed situation is unknown and hard to know. Assume an example based on independent recovery probability, fault f_i causes 50% errors in a system, and recovery action R_i has 80% probability to recover fault f_i and 0% probability to recover other faults; fault f_j causes other 50% errors in the system, and every other recovery action R_j has 0% probability to recover fault f_i;

assume there are 100 errors in the system, if R_i is selected in the first round, then 40 errors with fault f_i can be recovered, and the R_i probability of success in the first round attempts is 40%; there are 60 errors left, 10 with fault f_i and 50 with fault f_j, and R_i can recover 8 out of those 10 errors with fault f_i in second round, so the R_i probability of success is 8/60 (13.3%) in the second round attempts. The effect of recovery action on the second attempt will become more complex in complicated relationships between faults and recovery actions even though it is under the independent probability situation. The dependent probability situation is further complicated due to unknown dependency and unknown probability changes. However, we will consider the non-deterministic cases in our policy implementations.

This generic cost-based model to select the recovery action is based on the minimum total estimated recovery cost, which includes both the success case and fail case. The model also requires recovery-action success probability data and recovery cost data. We now address the problem of how to acquire this data.

3.3 Recovery Probability and Cost Observation

Our generic cost-based model is built on the recovery action success probability and recovery action cost. How these values can be collected? We created an observation model based on the Autopilot three-state recovery mechanism.

3.3.1 Recovery-Probability Observation

In the Autopilot recovery system, when an error is reported, the underlying fault is unknown; thus, we are not able to collect data on the probability of the recovery action successfully fixing a particular fault. However, we can track At_{Ri} how many times we attempt each recovery action Ri to fix errors in the system and As_{Ri} how many attempts it succeeds. Then, the probability that a given recovery action is successful in the system is:

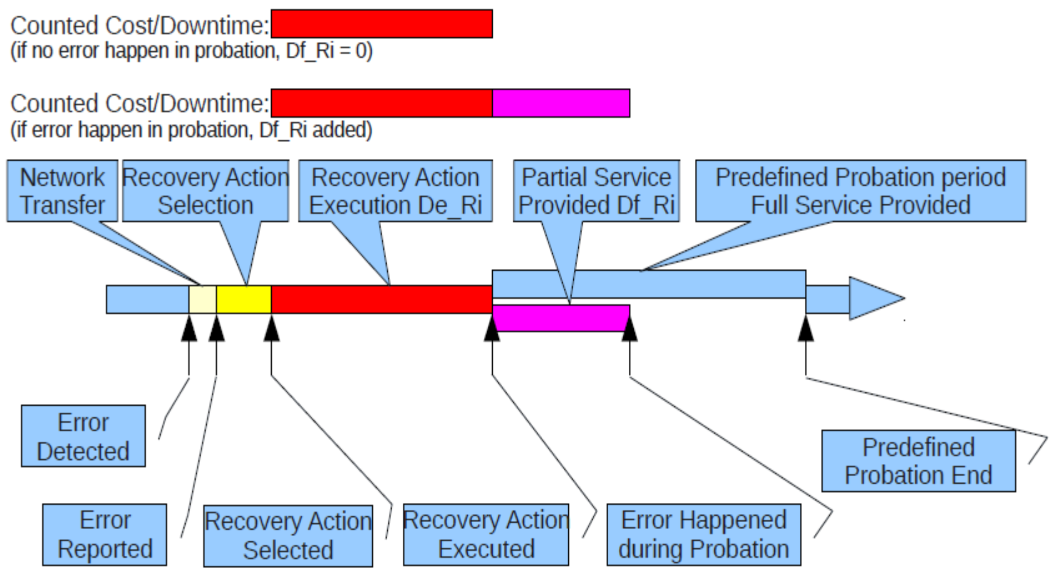

Counted Cost/Downtime:
(if no error happen in probation, Df_Ri = 0)
Counted Cost/Downtime:
(if error happen in probation, Df_Ri added)
Network Transfer
Recovery Action Selection
Recovery Action Execution De_Ri
Partial Service Provided Df_Ri
Predefined Probation period Full Service Provided
Error Detected
Error Reported
Recovery Action Selected
Recovery Action Executed
Error Happened during Probation
Predefined Probation End

Figure 3.2: Recovery Action Downtime

$$P_{Ri} = \frac{As_{Ri}}{At_{Ri}}$$

3.3.2 Recovery-Cost Observation

What is included in service downtime? We assume every service error is detected within a small time period in the Autopilot recovery system. Starting from detection, the error is reported to the central management service; a recovery action will be selected and executed; after the execution is finished, the service is in a probation state. From this process, we notice there is an error-reporting period, a recovery-action selection and execution period, and a probation period, during which we do not know if the recovery is successful or not (see Figure 3.2).

The error reporting and recovery-action selection period are likely negligible and so we ignore them. If this was not the case, their cost can be absorbed by the cost of the recovery action execution and probation period, as discussed below.

The recovery action execution period is potentially significant. Different recovery actions may have different costs. Further, the use of different recovery actions is a direct consequence of the selection of different automatic recovery policies. We will capture the recovery action execution period as the service downtime De_{Ri} .

The last cost is probation. During probation, an error may happen that causes the server to return to the failure state; if this happens, it means the previously selected recovery action did not fix the error and the server is assumed to be still down. This duration is treated a service downtime. We count duration Df_{Ri} from the time a recovery action has been executed and server is in probation to the time an error happens while still in probation. In such a case, we deem that the recovery action has failed and the cost of attempting the recovery action was $Cf_{Ri}=De_{Ri}+Df_{Ri}$. Conversely, the recovery action succeeded and the system has no additional error during the probation period, we deem that the recovery action was successful and had a cost of $Cs_{Ri}=De_{Ri}$.

We define $Af_{Ri}=At_{Ri}-As_{Ri}$ as the number of times Ri fails. Therefore, we can calculate the average cost of using recovery action Ri, which we call the recovery action unit cost UC_{Ri} as:

$$UC_{Ri}=\frac{\sum_{m=0}^{As_{Ri}} Cs_{Ri}(m)+\sum_{n=0}^{Af_{Ri}} Cf_{Ri}(n)}{At_{Ri}}$$

In this case, we can simply track and calculate recovery action unit cost instead of tracking and calculating both success unit cost and fail unit cost.

3.4 Final Recovery Action

There is a problem with our generic cost-based model: it always has certain probability that a recovery action could fail, so it always needs other recovery actions to finish the failed recovery work. Even if all recovery actions are used, there is still a chance of failure. This makes the model calculation incomplete.

In reality, the problem could always be resolved by an operator and other second line or even third line support people. After their thorough analysis, they may make a patch to the system, or replace some hardware components, even replace the whole server. In our recovery-action selection policy, human recovery is still a choice but it usually is the last choice, and it can always fix any problems, so we define it as “Final Recovery Action”.

The Final Recovery Action has the following characteristics:

- It always succeeds in fixing any error (P=1.0).

- It has highest unit cost.

- Human involvement in the recovery action is assumed.

- It could terminate the recursive selection calculation early, depending on the cost associated with the action.

Because the final recovery action will be a choice in each recovery action selection chain, it makes the selection complete with no remaining chance of failure. After the final recovery action is applied, because it is deemed successful, no probation period is assigned for it. The system moves to healthy immediately. Any error which happens after the applying final recovery action is deemed a new error.

3.5 Programing Algorithm Representation

Our generic cost-based recovery action selection model can be defined with the following pseudo code, which performance a depth first search over the recovery-action selection tree.

```
Cost_And_Action      minimal_Estimated_Total_Cost (Candidate_Actions set A)
{
    Cost_And_Action      result;
    result.cost = 0;

    while (set A is not empty)
    {
        Ri = retrieve next item from set A;

        UC_Ri = lookup_UnitCost (Ri);

        P_Ri = lookup_RecoverySuccessProbability (Ri);

        set B = remove Ri from set A;

        if (result.action is empty OR UC_Ri < result.cost)
        {
            if (P_Ri == 1)
                mETC_Ri = UC_Ri;
            else
                mETC_Ri=UC_Ri+(1-
    P_Ri)*minimal_Estimated_Total_Cost(set B).cost;

            if (result.action is empty OR mETC_Ri < result.cost)
            {
                result.action = Ri;
                result.cost = mETC_Ri;
            }
        }
    } /while

    Return result;
}
```

The recursive minimal_Estimated_Total_Cost function takes the passed in candidate recovery action set A and returns the result that contains the recovery action with associated minimal estimated total cost. The unit cost and recovery success probability lookup functions look up data from a table which is updated when a recovery action is executed and its consequence is known. The judgment of unit cost UC_Ri and recovery success probability P_Ri could end the recursive calculation earlier thus making it more efficient. However, further improving search performance is out of our scope.

3.6 Policy Implementation

We define different policies to implement the generic cost-based model in different ways. The difference is focused on different considerations when the previous recovery action is failed. When the recovery action failed, an error will be reported. We assume this is a repeating error and is caused by the same fault as the previous error. While we do not know exactly which fault is, we assume it is the same fault as caused the previous error and we know which recovery action was applied for that error and that it failed. Based on this information, we are able to adjust the probability of the previously used recovery action.

However, as required by estimated total cost model, policy P2, P3 and P4 all implement the following basic functions:

- Recording the recovery action success attempts and total attempts
- Calculating and updating the recovery action success probability whenever there is a change
- Recording the recovery action accumulated cost
- Calculating and updating the recovery action unit cost whenever there is a change
- Calculation and selection based on the estimated total cost model from a candidate recovery action set and based on the latest recovery action unit cost and success probability.

While Policy P1 is the implementation of the Autopilot simple heuristic policy, we do not further elaborate it. It is merely used for comparison.

3.6.1 Policy P2

Policy P2 implements the cost-based model. However, Policy P2 tracks the recent recovery history for recovery actions and their results for repeat errors; makes adjustments to the candidate recovery action set accordingly. The used recovery actions for a repeat error are excluded from further attempts. This policy thus assumes deterministic faults. The adjustment process is as follows:

- First, find all recovery actions that have been tried and failed to fix the same repeating error.

- Excludes those recovery actions from the original set A to form a new set A.

- Apply formula with new set A to the new recovery-action selection.

Policy P2 makes the following assumptions:

- A recovery action has no chance to succeed in the second and more attempts for an error which is not fixed in the first attempt; i.e., the success probability after the first attempt is 0%.

- A recovery action success probability in the first attempt equals the overall average success probability of this recovery action.

3.6.2 Policy P3

Policy P3 is a basic implementation to directly use the formula of Minimum Estimated Total Cost (minETC) in Recovery Action Set A under all circumstances. It always uses the current system-wide recovery action success probability and no adjustment is made to it. It assumes non-deterministic faults but it does not distinguish the first attempt to use a recovery action from the second attempt, third attempt, etc. Hence, it never tracks the result of the recovery attempts to the same error.

In addition to assuming non-deterministic faults, Policy P2 assumes that a recovery action success probability for all is the same, and always equals the overall average success probability of this recovery action. This assumption is clearly false, but sets a baseline for comparison.

3.6.3 Policy P4

Policy P4 tracks the recent recovery history. Like Policy P3, it considers failed recovery actions for further attempts in cases of repeated errors; however, it lowers the success probability of any failed action for the further attempts. This more accurately takes into consideration non-deterministic faults. The adjustment process is as follows:

- First, find out the number of attempts for each recovery action which has been tried and has failed to fix the same repeating error.

- Reduce the system-wide success probability of those tried and failed recovery actions by multiplying by a coefficient ($0<X<1$); we use X^n, where n is the number of attempts, the more attempts made, the more the probability is reduced. Note that the adjusted probability is not written back to the system-wide success probability record; it is only used to calculate and select the recovery action under the multiple-attempts situation.

- Apply formula with set A and the adjusted recovery actions success probabilities to the new recovery action selection.

Policy P4 makes the following assumptions:

- A recovery action has less and less chance to recover the system in the second and subsequent attempts for the same error which is not fixed in the first attempt.

- A recovery action success probability in the first attempt equals the overall average success probability of this recovery action.

3.7 Example Illustration

We demonstrate with a virtual example to see how the policies based on minimum estimated total cost work and the different results from their different implementations. We also discuss the important initial value before the example walk through.

3.7.1 Initial Value Setup

There are some initial values which have to be setup to make a policy bootstrap and run properly. These values are located in three areas: defining recovery actions; recovery action cost table; recovery action success probability table.

In order to make the example clear and simple, we define four recovery actions: R1, R2, R3, and the final recovery action R4. As mentioned in section 3.4, the final recovery action is required to ensure the recovery action selection calculation terminates.

The next step is to setup the values in the recovery action cost table (see Table3.1) and probability table (see Table 3.2). The initial values should not only be able to bootstrap the algorithm, but also give a fair chance to all recovery actions. The initial success probability given to all recovery actions is 100% and the initial recovery action unit cost given to all actions is 0. After one recovery action is selected, the cost must be greater than 0, so no matter whether its probability is still 100% or not, it will not be the best choice compared to the untried recovery actions. By using these initial values, the algorithm will select each recovery action in the first four errors so that they each get an initial execution. Then, based on the observed recovery success rate and recovery action unit cost, the algorithm will continue select the lowest total recovery cost action.

The initial recovery action unit cost 0 and success probability 100% enable the bootstrap. What are reasonable values for the remaining variables? As the recovery action unit cost is 0, the accumulated recovery action cost must be 0. The total attempts could be any number; however, if the number other than 0 is set, the next time the unit cost calculation will be low.

Recovery Action	**Total Attempts**	**Accumulated Cost (seconds)**	**Unit Cost (seconds)**
R1	0	0	0
R2	0	0	0
R3	0	0	0
R4 (final)	0	0	0

Table 3.1: Initial Recovery Action Cost Table

Recovery Action	**Total Attempts**	**Success Attempts**	**Success Probability (%)**
R1	10	10	100
R2	10	10	100
R3	10	10	100
R4 (final)	10	10	100

Table 3.2: Initial Recovery Action Success Probability Table

There is a similar question for setting up the total attempts and success attempts values in recovery action success probability table. As the initial success probability is 100%, the total attempts and success attempts must be the same. As there is no observation yet, we could give both variables 0. It could bootstrap in the first step; however, in the first step if the recovery action failed, the total attempts becomes 1 and success attempts is still 0, and thus the success probability becomes 0%; as such, this recovery action will never be selected again. Similarly, we could set both variables to 1, then in the first step if the recovery action failed, the total attempts becomes 2 and success attempts is still 1; the success probability drops to 50%, meaning this recovery action may have much less chance to be selected. As a reasonable alternative, we initialize both variables to 10. In order to quickly drop to 50%, it must fail for 10 times in the first 10 attempts. In this case the probability will not drop too fast and give the model enough opportunity to learn each recovery action in a reasonable time frame. We investigate alternatives to this approach in Chapter 6.

3.7.2 Example Walk Through

We walk through the first six errors to see how the different policies work, trace their cost table and probability table changes, and calculate the selection to see different results. Policy P2, P3 and P4 each start from the same initial values. For the purpose of this example R1 is very cheap, and subsequent recovery actions are more expensive. Our algorithm does not know this.

First Error Detected

Because all policies have the same initial table values, and there is no previous failed recovery history, there is no adjustment made to Policy P2 and P4. All Policies calculate based on the basic model with the same values, and get the same result set:

$$\min_{R1\in A} ETC_{R1} = UC_{R1} + \min_{Rx\in A, Rx\neq R1} ETC_{Rx} * (1-P_{R1}) = 0 + \min_{Rx\in A, Rx\neq R1} ETC_{Rx} * (1-1) = 0$$

$$\min_{R2\in A} ETC_{R2} = UC_{R2} + \min_{Rx\in A, Rx\neq R2} ETC_{Rx} * (1-P_{R2}) = 0 + \min_{Rx\in A, Rx\neq R2} ETC_{Rx} * (1-1) = 0$$

$$\min_{R3\in A} ETC_{R3} = UC_{R3} + \min_{Rx\in A, Rx\neq R3} ETC_{Rx} * (1-P_{R3}) = 0 + \min_{Rx\in A, Rx\neq R3} ETC_{Rx} * (1-1) = 0$$

$$\min_{R4 \in A} ETC_{R4} = UC_{R4} + \min_{Rx \in A, Rx \neq R4} ETC_{Rx} * (1 - P_{R4}) \ = \ 0 + \min_{Rx \in A, Rx \neq R4} ETC_{Rx} * (1-1) = 0$$

Because all results are the same, and we simply select them in sequence, recovery action R1 is selected. R1 is then executed, and the server is back to running. The 10 seconds execution time of R1 is then added to R1's accumulated cost (AC), and R1's unit cost (UC) is then calculated. The cost table for all policies changes to:

Recovery Action	**Total Attempts**	**Accumulated Cost (seconds)**	**Unit Cost (seconds)**
R1	1	10	10
R2	0	0	0
R3	0	0	0
R4 (final)	0	0	0

Second Error Detected

After the server is back to running for just 20 seconds, a second error is reported. It is then assumed that the previous recovery action R1 failed to fix the first error. Thus 20 seconds downtime is counted to recovery action R1's cost. The cost table and success probability table are updated accordingly for all policies:

Recovery Action	**Total Attempts**	**Accumulated Cost (seconds)**	**Unit Cost (seconds)**
R1	1	30	30
R2	0	0	0
R3	0	0	0
R4 (final)	0	0	0

Recovery Action	**Total Attempts**	**Success Attempts**	**Success Probability (%)**
R1	11	10	90.9
R2	10	10	100
R3	10	10	100
R4 (final)	10	10	100

Policy P3 does the same estimated total cost calculation and selection based on these new values. R2, R3, R4 are still 0, so R2 is selected by P3. Because recovery action R1 was failed to fix the previous error, Policy P2 removes R1 from the candidate recovery actions, and does the calculation and selection from the rest; the same result as P3 is produced: R2 is selected by P2. Policy P4 first lowers the R1 success probability to (90.9% * 0.5 = 45.45%), then does calculation and selection based on the adjusted value; the same result as P2 and P3 is produced: R2 is selected by P4. R2 is then executed for all policies; the server is back to running. The 50 seconds execution time of R2 is then added to R2's accumulated cost (AC), and R1's unit cost (UC) is then calculated. So the cost table for all policies changes to:

Recovery Action	**Total Attempts**	**Accumulated Cost (seconds)**	**Unit Cost (seconds)**
R1	1	30	30
R2	1	50	50
R3	0	0	0
R4 (final)	0	0	0

After two hours, the predefined probation period, there is no error detected. So server is assumed healthy. Policy P2 and P4 clear their recent recovery history. The recovery action success probability table is updated accordingly for all policies:

Recovery Action	**Total Attempts**	**Success Attempts**	**Success Probability (%)**
R1	11	10	90.9
R2	11	11	100
R3	10	10	100
R4 (final)	10	10	100

Third Error Detected

After a long period a third error is detected. According to the latest values in cost and probability tables, after calculation R3 and R4 both cost 0, and so R3 is selected by all policies. R3 is then executed; the server is back to running. The 600 seconds execution time of R3 is then added to R3's accumulated cost (AC), and R3's unit cost (UC) is then calculated. So the cost table for all policies changes to:

Recovery Action	**Total Attempts**	**Accumulated Cost (seconds)**	**Unit Cost (seconds)**
R1	1	30	30
R2	1	50	50
R3	1	600	600
R4 (final)	0	0	0

After the two hours predefined probation period there is no error detected, so the server is assumed healthy. Policy P2 and P4 clear their recent recovery history. The recovery action success probability table is updated accordingly for all policies:

Recovery Action	Total Attempts	Success Attempts	Success Probability (%)
R1	11	10	90.9
R2	11	11	100
R3	11	11	100
R4 (final)	10	10	100

Fourth Error Detected

R4 is selected by all policies as it is the only action currently without cost. No error happens in probation, so the cost and probability tables for all policies are updated to:

Recovery Action	Total Attempts	Accumulated Cost (seconds)	Unit Cost (seconds)
R1	1	30	30
R2	1	50	50
R3	1	600	600
R4 (final)	1	7200	7200

Recovery Action	Total Attempts	Success Attempts	Success Probability (%)
R1	11	10	90.9
R2	11	11	100
R3	11	11	100
R4 (final)	11	11	100

Fifth Error Detected

After a long period, the fifth error is detected; since there is no recent failed-recovery history, all policies do the same calculation and selection based on the same table values.

$$\min_{R1\in A} ETC_{R1} = UC_{R1} + \min_{Rx\in A, Rx\neq R1} ETC_{Rx} * (1 - P_{R1}) \;=\; 30 + \min_{Rx\in A, Rx\neq R1} ETC_{Rx} * (1 - 0.909) = 34.5$$

$$\min_{R2\in A} ETC_{R2} = UC_{R2} + \min_{Rx\in A, Rx\neq R2} ETC_{Rx} * (1 - P_{R2}) \;=\; 50 + \min_{Rx\in A, Rx\neq R2} ETC_{Rx} * (1 - 1) = 50$$

$$\min_{R3\in A} ETC_{R3} = UC_{R3} + \min_{Rx\in A, Rx\neq R3} ETC_{Rx} * (1 - P_{R3}) \;=\; 600 + \min_{Rx\in A, Rx\neq R3} ETC_{Rx} * (1 - 1) = 600$$

$$\min_{R4\in A} ETC_{R4} = UC_{R4} + \min_{Rx\in A, Rx\neq R4} ETC_{Rx} * (1 - P_{R4}) \;=\; 7200 + \min_{Rx\in A, Rx\neq R4} ETC_{Rx} * (1 - 1) = 7200$$

Recovery action R1 is selected by all policies; R1 is then executed, and the server is back to running. The 12 seconds execution time of R1 is then added to R1's accumulated cost (AC), and R1's unit cost (UC) is then calculated. So the cost table for all policies changes to:

Recovery Action	**Total Attempts**	**Accumulated Cost (seconds)**	**Unit Cost (seconds)**
R1	2	42	21
R2	1	50	50
R3	1	600	600
R4 (final)	1	7200	7200

Sixth Error Detected

After server is back to running for just 28 seconds, an error is reported. It is assumed the previous recovery action R1 failed to fix the fifth error. Thus 28 seconds downtime is counted to recovery action R1's cost. The cost table and success probability table are updated accordingly for all policies:

Recovery Action	**Total Attempts**	**Accumulated Cost (seconds)**	**Unit Cost (seconds)**
R1	2	70	35
R2	1	50	50
R3	1	600	600
R4 (final)	1	7200	7200

Recovery Action	**Total Attempts**	**Success Attempts**	**Success Probability (%)**
R1	12	10	83.3
R2	11	11	100
R3	11	11	100
R4 (final)	11	11	100

Each policy does the same estimated total cost calculation and selection based on these new values. R2, R3, and R4 calculation results have no change; they are still 50, 600, and 7200, respectively. However, R1 changes:

$$\min_{R1\in A} ETC_{R1} = UC_{R1} + \min_{Rx\in A, Rx\neq R1} ETC_{Rx} * (1 - P_{R1}) = 35 + \min_{Rx\in A, Rx\neq R1} ETC_{Rx} * (1 - 0.833) = 43.3$$

R1 is selected by P3. Policy P2, by contrast, remove R1 from consideration because it failed to fix the previous error. Therefore, R2 is selected by Policy P2. Policy P4 first lowers the R1 success probability to (83.3% * 0.5 = 41.65%), and then does a calculation and selection based on the adjusted value:

$$\min_{R1\in A} ETC_{R1} = UC_{R1} + \min_{Rx\in A, Rx\neq R1} ETC_{Rx} * (1 - P_{R1}) = 35 + \min_{Rx\in A, Rx\neq R1} ETC_{Rx} * (1 - 0.4165) = 64.17$$

R2 is 50, so R2 has minimum estimated total cost, thus R2 is also selected by P4. After R1 is executed by Policy P3 and R2 executed by Policy P2 and P4, the respective cost and probability tables for each policy will be updated accordingly. This example demonstrates how different policies use the estimated total cost model in different approaches, and how the initial value setup is used.

Chapter 4 Simulator Design

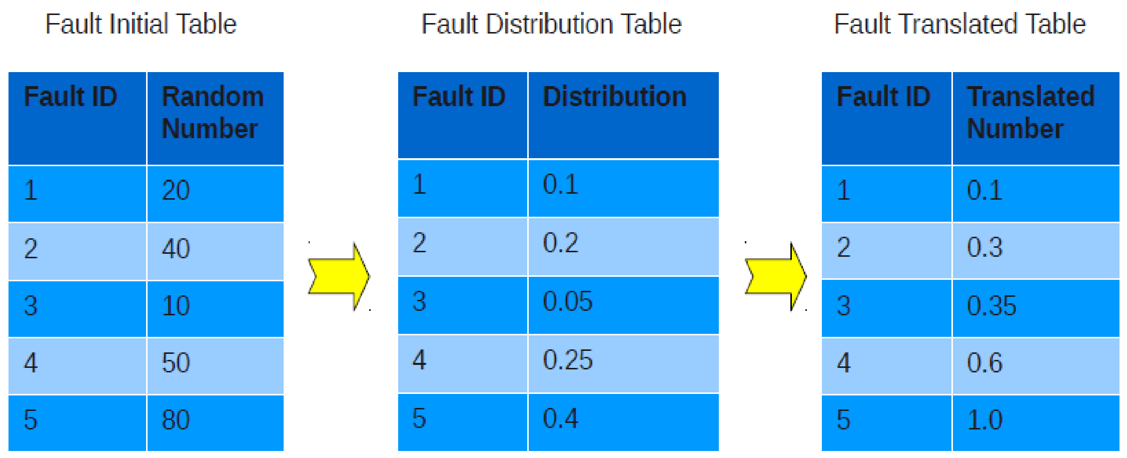
In this chapter, we describe different simulation test data, including fault and error data, the effect of recovery actions on faults, and recovery action execution times. We then illustrate our simulator implementation, the different components, and how they work together.

4.1 Simulation Test Data

Reasonable test data generation is key in a simulation test environment. We carefully examined the various data and how it should be generated. First we looked at how errors should be generated, what the relationship between errors and faults are, and what a reasonable error interval is. Then we studied what recovery action data is needed and how it should be generated, and what the relationship is between recovery actions and faults.

4.1.1 Fault and Error

In the real data center many errors happen. After analyzing those errors, find the faults behind them and fixing them, a report can usually show the percentage of each fault type. Each data center has different environments and different types of employee, they usually have different fault distribution as well. Before we generate any errors for our simulation test, we first generate a fault distribution for each run of the test (See Figure 4.1). Based on maximum faults predefined in test configuration file, a pseudo-random number generator with a passed-in seed generates a random number between 0 to 100 for each fault. The number for each fault is then summed up to get the total. The number for each fault will finally be divided by the total to get the percentage of each fault among total faults. After each fault gets its percentage, a translated fault distribution table will be created in order to facilitate the error generation. As fault ID followed one by one, the translated percentage of each fault is its original percentage plus all percentage numbers from prior faults.



Fault Initial Table

Fault ID	Random Number
1	20
2	40
3	10
4	50
5	80

Fault Distribution Table

Fault ID	Distribution
1	0.1
2	0.2
3	0.05
4	0.25
5	0.4

Fault Translated Table

Fault ID	Translated Number
1	0.1
2	0.3
3	0.35
4	0.6
5	1.0

Figure 4.1: Fault Distribution Table

During the simulation testing, the error is generated in real time based on fault distribution data. When an error is needed to be generated in our simulator, a fault is also assigned. A pseudo-random number generator with a passed-in seed generates a random number between 0 and 1. This number is used to look up the translated fault table starting from the first fault ID which has the smallest translated percentage number. Along the fault ID increasing, the percentage number is also increased. By following the fault ID, a fault will eventually shows a bigger translated percentage number than the random number, that fault is now assigned to the error. As long as enough errors can be generated, this approach can guarantee those errors' fault following the generated fault distribution, meanwhile any error's specific fault is not predictable.

After recovery action is applied to an error, our simulator implementation will know exactly whether the previous error has been fixed. If the previous error was fixed successfully, the automatic recovery policy will judge the server to have passed from the probation state into the Health state, which means no new error in probation period. However, in real environment a fresh new error of any fault type could happen at any time, it should not be based on policy defined probation period. When we simulate the error interval,

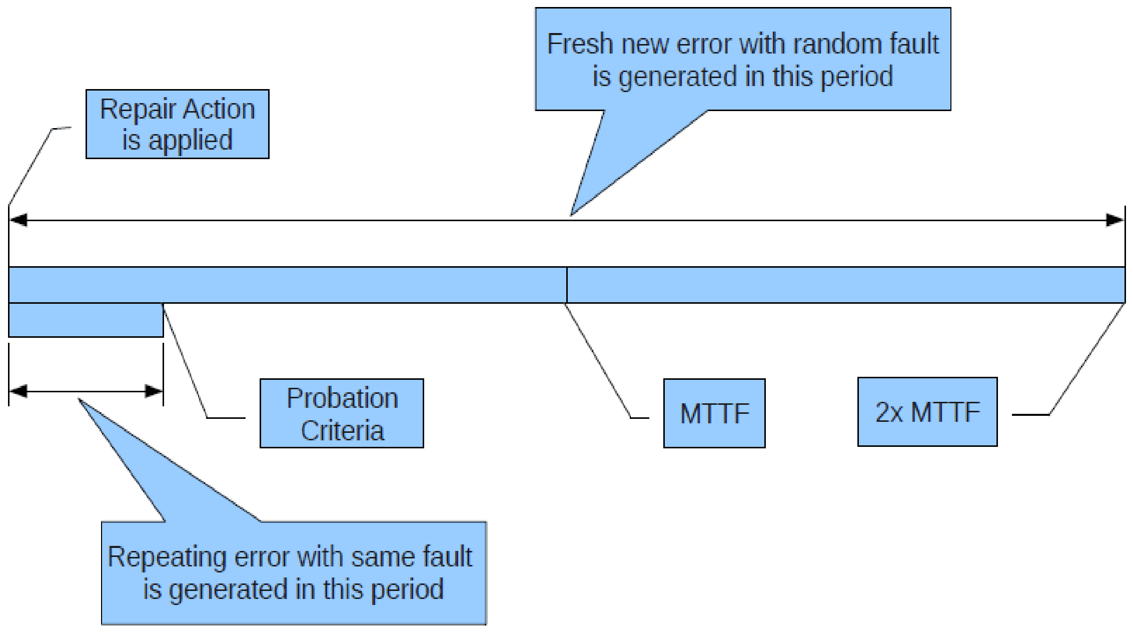

Fresh new error with random fault is generated in this period
Repair Action is applied
Probation Criteria
MTTF
2x MTTF
Repeating error with same fault is generated in this period

Figure 4.2: Repeating Error and Fresh New Error

we predefined an maximum server health time in configuration file. The maximum health time is set as double of the average health time, which can be considered similarly to the MTTF mean time to failure. Based on above consideration, a reasonable maximum health time can be set and adjusted to a similar known data center characteristic. Thus the random fresh new error interval between 0 and maximum health time is generated from a pseudo-random number generator. The fresh new error with random fault will then be set to happen after the random error interval elapsed from the time previous recovery action was executed (see Figure 4.2).

When a recovery action is applied to fix an error but failed, usually we can see a similar error happened shortly after the recovery action is applied in real world, and the similar error should have the same fault which was not fixed. What that short period could be? It is difficult to know. This is also the question from the policy judgement of recovery success. Since both simulation and policy implementation are looking for the same answer, we just align them together, so the short period for repeating error in both cases will be probation period. A random shorter error time interval is then generated from a pseudo-random number generator which generates a random number between 1 second and probation period threshold.

This simulated error and fault generation also minimizes the misjudgement impact caused from improperly defined policy probation period. Because the simulated repeating error is generated exactly within the defined probation period by which the policy judges if the recovery success - if probation period is defined as two hours, the repeating error will be generated within two hours, and so on. So probation period definition does no longer have significant impact to our research. However in reality, the fresh new error may happen within probation and the repeating error may happen after probation. This is also achieved in our simulation. Because the generated new error could happen in any arbitrary time, it may fall in probation period. Also because the generated new error could be of arbitrary fault type, it may have the repeating fault just after probation period. This greatly mimics the real environment which causes imperfect judgement of recovery success in real policy implementation.

4.1.2 Recovery Action to Fault Effects

As each error has real fault behind, which reflects the root cause behind the error. In real world, the recovery action applied to each error also has result – fixed or not fixed, that reflects the effect of the recovery action to the fault behind the error. Following the real world mechanism, the test data of the recovery action effect to each fault is generated before each test. Also as observed from real world , a recovery action does not have average effect to all faults. A recovery action is usually designed to target certain faults and not for all faults. For the targeted faults the recovery action usually has very high effect, but minimal effect for the rest faults. Thus we applied a 80/20 rule to generate the recovery action effects. For a given recovery action, the effect to targeted faults is higher than 80% chance to success, and the effect for rest faults is lower than 20% chance to success.

The recovery action effects test data generation uses three pseudo-random number

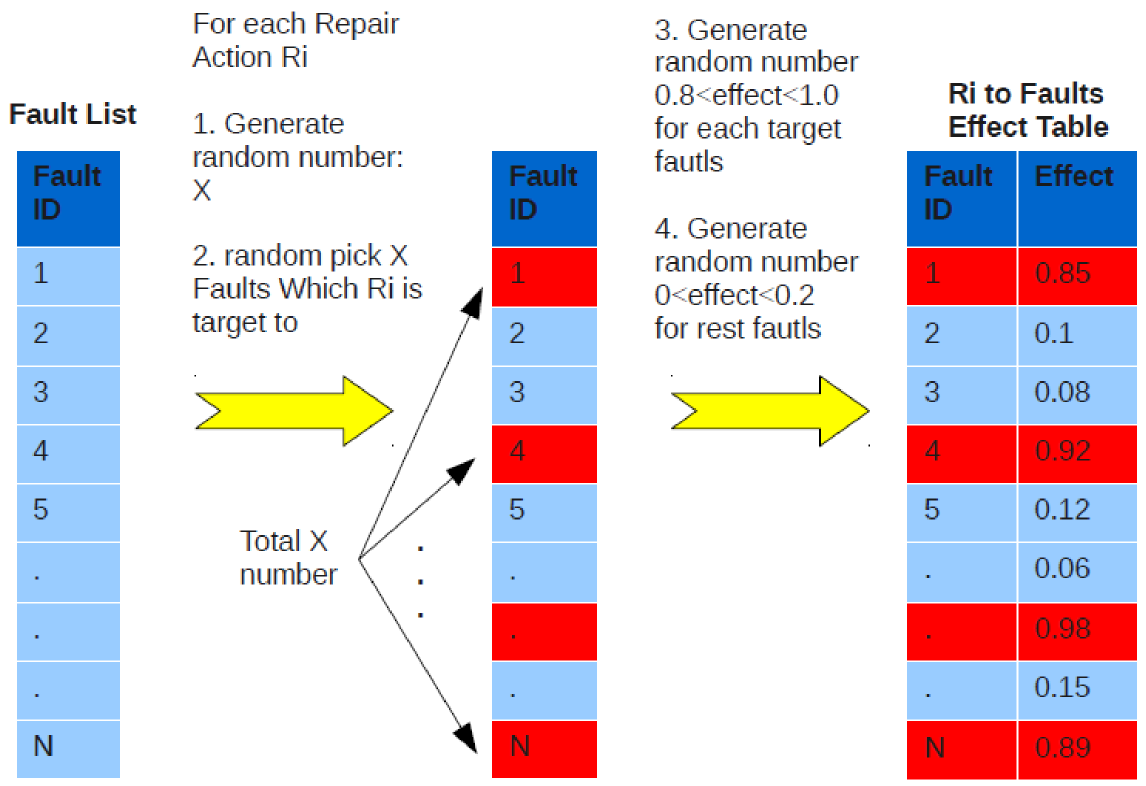

Fault List
Fault ID
1
2
3
4
5
.
.
.
N
For each Repair Action Ri
1. Generate random number: X
2. random pick X Faults Which Ri is target to
Total X number
Fault ID
1
2
3
4
5
.
.
.
N
3. Generate random number 0.8<effect<1.0 for each target fautls
4. Generate random number 0<effect<0.2 for rest fautls
Ri to Faults Effect Table
Fault ID
Effect
1
0.85
2
0.1
3
0.08
4
0.92
5
0.12
.
0.06
.
0.98
.
0.15
N
0.89

Figure 4.3: Repair Action to Fault Effects

generators (see Figure 4.3). It uses the first pseudo-random number generator to generate a random number of how many faults this recovery action will target to, which is a number between 0 and the predefined maximum faults in configuration file. It then runs in a loop for number of target faults, and uses the second pseudo-random number generator to generate the random targeted fault IDs. It finally uses the third pseudo-random number generator to generate a random number between 0.8 and 1 for selected target faults and 0 to 0.2 for the rest faults. After these steps, the recovery action gets effect probability for each fault. By repeating those processes for each recovery action, all recovery actions get effect probability for all faults.

These pre-generated recovery action to fault effects data before each test running is then used by simulator to generate real time recovery success or fail effects. Because it is simulated environment, the simulator knows exact fault behind each error. After a recovery action Ri is applied to an error, simulator will look for the recovery action Ri to faults effect table and find the effect probability EP(Ri, F) to the fault F which is behind the applied error. Then simulator will generate a random number between 0 and 1. At last the simulator will compare the random number with EP(Ri, F): if the random number is less than EP(Ri, F) – it falls in the success probability and the error should be fixed, a fresh new error with arbitrary fault and random period will be generated; if the random number is greater than EP(Ri, F) – it is outside the success probability and the error should not be fixed, a repeating error with same fault F and random shorter period within probation will be generated.

4.1.3 Recovery Action Execution Time

Another important recovery action data to be generated is the recovery action execution time. As observed from real world, a same recovery action may take different lengths when it is executed every time. However different recovery actions have their own range of the recovery action execution time. For example, the restart recovery action may take 10 seconds in one recovery attempt, sometimes it takes 6 seconds or 20 seconds in other attempts, it however would not take 10 hours, a day or two. The human trouble shooting may take 2 hours in one recovery attempt, sometimes it takes half hour or 48 hours, it would never take couple seconds to fix an error. Our recovery action execution time generation hence accommodates these two characters - each recovery action has its own range of recovery execution time and each time the recovery action take different execution time within its range.

The recovery action execution time is generated in two phases (see Figure 4.4). Before each test run, the base recovery execution time is generated for each recovery action by a pseudo-random number generator, which generates the random number between minimum and maximum base recovery execution time predefined in the configuration file. The final recovery action base execution time is predefined and retrieved from configuration file directly. With maximum recovery execution time and the final recovery action base execution time defined appropriately, final recovery action can make proper distance from

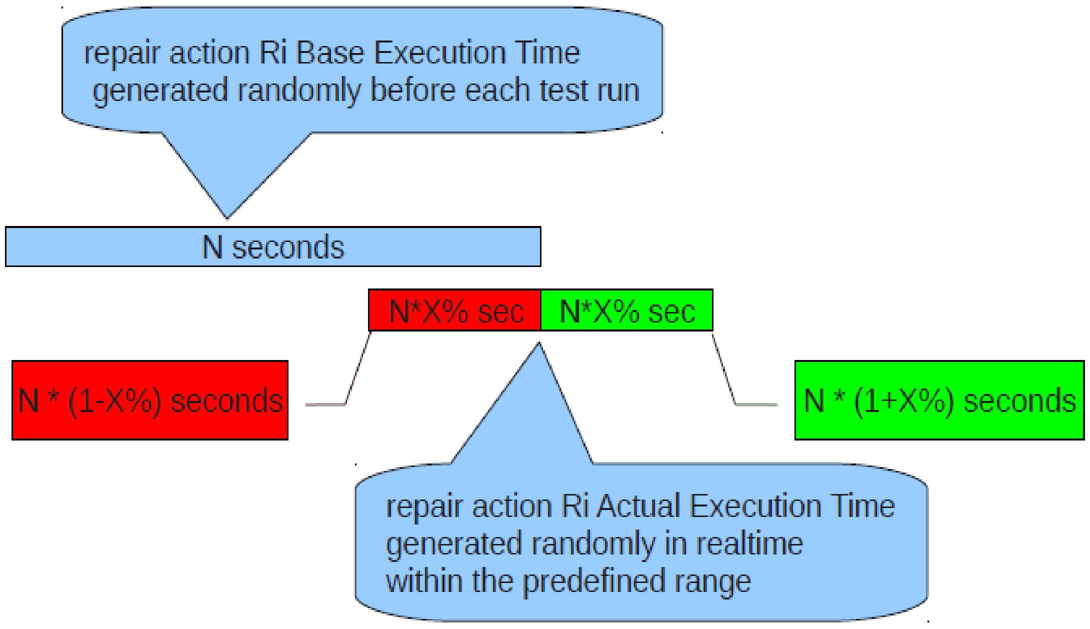

repair action Ri Base Execution Time generated randomly before each test run
N seconds
N*X% sec
N*X% sec
N * (1-X%) seconds
N * (1+X%) seconds
repair action Ri Actual Execution Time generated randomly in realtime within the predefined range

Figure 4.4: Base and Actual Execution Time

other recovery action base execution time. During the test running, every time a recovery action is executed, its actual execution time is randomly generated by a pseudo-random number generator within a predefined floating range around its base execution time.

4.2 Simulator Implementation

As mentioned in Background chapter, due to the restricted situation, we test our policy model in simulation environment. And we implement a similar Autopilot data center environment with discrete event simulation. We describe different components in our simulator implementation (see Figure 4.5), from the different event type to system state component, then the recovery action selector component, finally the core component – simulation controller, in which we also illustrate how it process different events and how it works with other components.

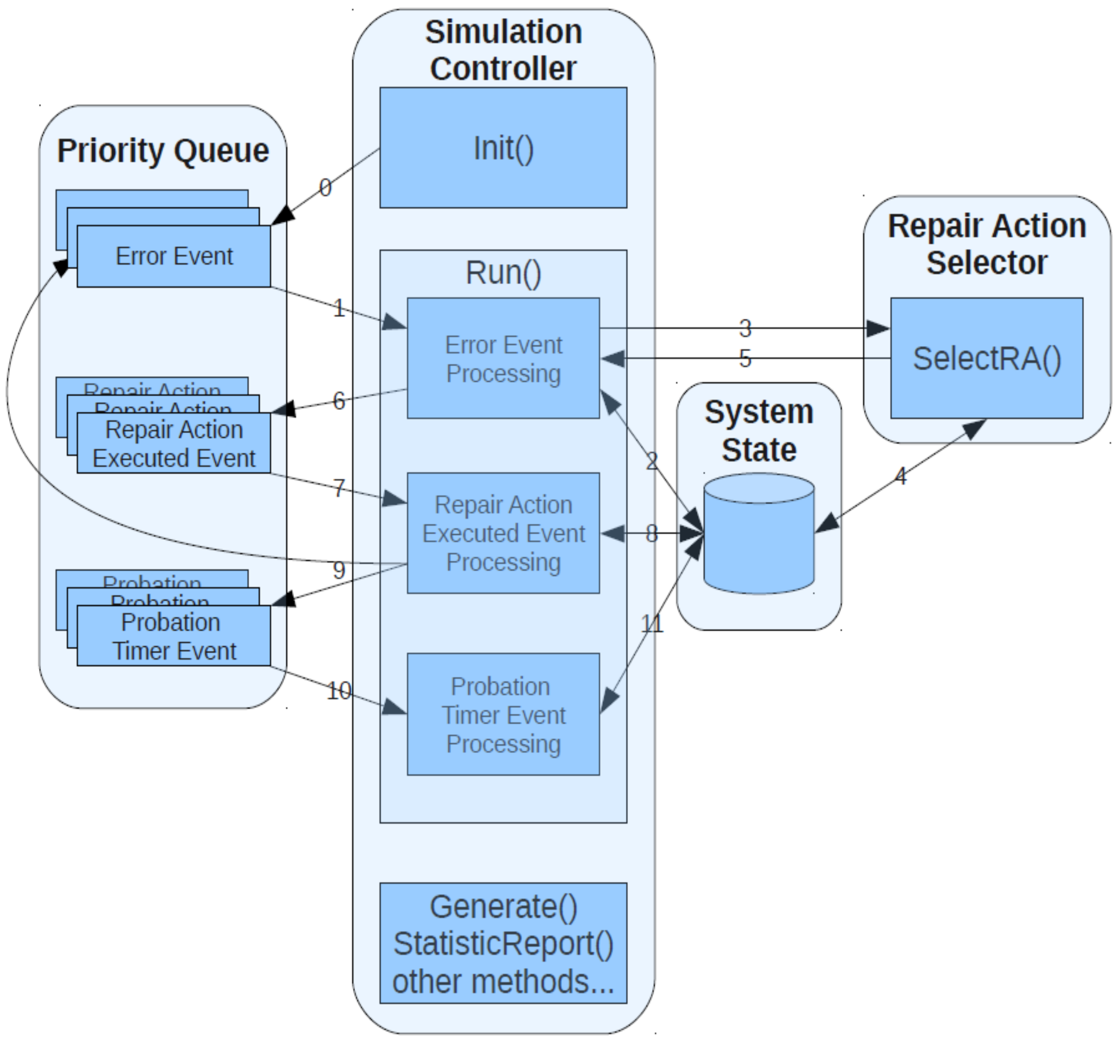


Figure 4.5: Simulator Implementation

4.2.1 Event Types

To simulate different events in data center, we defined three types of events in the simulator: Error Event, Recovery Action Executed Event and Probation Timer Event. Other than the most essential field – event time, all event types also contain Server ID as they are all server dependent events.

- Error Event - As described in Autopilot, whenever an error is detected on a server, the error is reported to device manager. The error event mimics that reported error. A special field – Fault is also defined in error event. This field is not contained in the real data center error report, neither used by any recovery action selection policy. This is solely used for the simulator to generate recovery success or fail effect (see section 4.1.2).

- Recovery Action Executed Event - After recovery action is selected and applied, the sever notices device manager that it is done. We thus defined it as the Recovery Action Executed Event. Also for simulation purpose, this event contains the executed recovery action, error happen time and fault.

- Probation Timer Event - The device manager will set the server from probation to health if no error happens in probation period. We defined it as the Probation Timer Event to deal with this scenario.

4.2.2 System State

The system state component stores four system state related tables, and these state tables are kept updated along the system running.

- Server State Table - This table is used to track each server status, whether they are in Health, Failure or Probation state and when the state is changed. This table contains server ID as the key and server state and state change time as the value.

- Recent Server Recovery History - This table is used by certain policies to record and update the recent recovery action history for each server, it is used as a tool to understand what recovery actions have been applied before a server returns to health state. This table contains server ID as the key and a series of recently used recovery actions for the specified server as the value.

- Recovery Action Cost Table - This table is used by our policy to track the cost for each recovery action. (see section 3.7 for details)

- Recovery Action Success Probability Table - This table is used by our policy to track the success probability for each recovery action. (see section 3.7 for details)

4.2.3 Recovery Action Selector

This Recovery Action Selector component is implemented by different policies. This component defines the only function selectRA(), which needs to return a selected recovery action for a specified server. Different policies implement this function with their own algorithm. This structure makes it easily extend to any new policy implementations.

4.2.4 Simulation Controller

This is the core discrete event simulation processing component. We implemented the typical init(), run() and statisticReport() functions. We also implemented generate() and other helper functions.

- generate() - This function generates three static reference tables and associated data. Those tables contain fault distribution data, recovery action effect to each fault, and recovery action base execution time data.

- init() - This function first initializes system state which contains server state table, server recovery action history table, recovery action cost table, recovery action

```
public void run(RepairActionSelector repairActionSelector)
{
        while (PriorityQueue is not empty && errorCounter <= maxTestErrors)
        {
        CurrentEvent = PriorityQueue.poll();
        CurrentServerID = CurrentEvent.getServerID();
        CurrentEventTime = CurrentEvent.getTime();

        if (CurrentEvent is ErrorEvent)
        {
                  ...
                }else if (CurrentEvent is RepairActionExecutedEvent)
                {
                  ...
                }else if (CurrentEvent is ProbationTimerEvent)
                {
                  ....
                }
         }
}
```

Figure 4.6: run() Pseudo Code

success probability table. This function then sets or resets all counters and statistic report data to initial values. Next, this function instantiates all pseudo-random number generators and sets corresponding seeds generated from generate() function. Finally a priority queue is instantiated, initial error events are generated based on predefined number of servers and put into priority queue for further processing.

- run() - This is the event processing function. (see Figure 4.6) The major structure is a while loop which checks if there is any event in the priority queue. If there is no event in the priority queue, the while loop is ended and the run() function is also finished. If there is event in the priority queue, the event with earliest event time will be returned by the priority queue. Based on different event types, the event will then be processed by different logic blocks within the while loop.

```
if (CurrentEvent is ErrorEvent)
{
        Get CurrentServerState from ServerState Table for CurrentServerID;

        if (CurrentServerState is Health)
        {
        find previous RA from RA history in ServerRAHistory Table;

        //update RA probability Table with:
                totalAttempts+1;
                probability = successAttempts / totalAttempts;

              //update RA cost Table with:
                accumulatedCost += probation duration;
                unit cost = accumulatedCost / totalAttempts;
        }
        // update ServerState Table with:
           server state = Failure;
           state time = current event time;

        SelectedRA = RepairActionSelector.selectRA(serverID);

        raee = new RepairActionExecutedEvent ;
        raee.AppliedRA(SelectedRA);
        raee.EventTime = CurrentEventTime+ Random RAExecutionTime;
               ...

        insert RepairActionAppliedEvent to PriorityQueue;
}
```

Figure 4.7: ErrorEvent Processing Pseudo Code

If next event is Error Event (see Figure 4.7), the process does the following: firstly it finds current server status from system state; secondly it checks if server is not in Health which means previous recovery action was failed, it then updates success probability table and cost table for previous recovery action (it finds the previous recovery action from Recent Server Recovery History table; the recovery action success probability will be recalculated and updated by increasing the total

```
else if (CurrentEvent is RepairActionAppliedEvent)
{
        // update ServerState Table with:
           server state = Probation;
           state time = current event time;

        Add applied repair action in RecentServerRepairHistory table;

        RepairExecutionTime = CurrentEventTime - ErrorHappenTime;
        //update RepairAction cost table with:
         accumulatedCost += RepairExecutionTime;
         totalAttempts+=1;
         unit cost = accumulatedCost / totalAttempts;

        Generate random number between 0 and 1.
        Get RepairActionEffectProbability(applied repair action, error fault);
        if (random number < RepairActionEffectProbability)
        {
        nextErrorFault = generate random fault;
        nextErrorPeriod = generate random error period;
        }else
        {
        nextErrorFault = the current error fault type;
        nextErrorPeriod = generate random short error period;
        }
        Generate new ErrorEvent (next Error Fault, next Error Period);
        insert ErrorEvent to PriorityQueue;

        Generate new ProbationTimerEvent PTE;
        PTE.EventTime = CurrentEventTime + probationThreshold;
        insert ProbationCheckerEvent to PriorityQueue;
}
```

Figure 4.8: RepairActionExecutedEvent Processing Pseudo Code

attempts but not the success attempts; the recovery action unit cost will also be recalculated and updated by adding the probation additional cost.); thirdly it changes server state to Failure and set state time to current event time; next it computes preferred recovery action from Recovery Action Selector; finally it generates a future

```
else if (CurrentEvent is ProbationCheckerEvent)
{
        Get CurrentServerState and CurrentStateTime from ServerState Table for CurrentServerID;

        if (CurrentServerState is Probation
           && (CurrentEventTime - CurrentStateTime) == probationThreshold)
        {
            // update ServerState Table with:
               server state = Health;
               state time = current event time;

             //update RA probability Table with:
               totalAttempts++;
               success++;
               probability = successAttempts / totalAttempts;

             clear repair action history in ServerRepairActionHistory Table;
        }
}
```

Figure 4.9: ProbationTimerEvent Processing Pseudo Code

Recovery Action Executed Event which is set to current event time plus a random recovery action execution time and put the event into Priority Queue.

If next Event is Recovery Action Executed Event(see Figure 4.8), the process does the following: firstly it changes the server state to Probation and sets the state time to current event time; secondly it records the executed recovery action to Recent Server Recovery History table; thirdly it computes recovery action execution time and updates recovery action cost table; next it determines if recovery action success or fail and generates next Error Event accordingly with proper fault type and Error Event time (see 4.2.2 for details); then it puts the next Error Event into Priority Queue; finally it generates a Probation Timer Event which is set to current event time plus the probation threshold and puts the event into Priority Queue.

If next Event is Probation Timer Event (see Figure 4.9), the process verifies if latest recovery action is success by checking two conditions: the server is in Probation and current probation checking event time subtract server state time equals probation threshold, that means no error happened in probation period namely recovery is deemed success. Then it does the following: it changes server state to Health and sets state time to current event time; the recovery action success probability will be recalculated and updated by increasing the total attempts and increasing the success attempts; it finally clears recent server recovery history.

To sum it up, the different event type processing usually updates server state for the event related server, records or clears the recent server recovery history table, updates recovery action cost table and recovery action success probability table accordingly, generates necessary future events and put into the priority queue. The recovery action cost is updated immediately after recovery action is applied in Recovery Action Executed Event processing, and gets further updated for additional cost if there is error happened during probation in Error Event processing. The recovery action success probability is updated when we know the recovery action is either succeed - checked in Probation Timer Event processing, or failed - checked in Error Event processing.

- statisticReport() - During the processing of events, whenever there is update to the recovery action cost table, the statistic total server cost is also updated. And the total server run-time is updated when the last error event is processed. When simulation running finishes, this function simply calculates the ratio of total server cost to total server run time and returns that ratio along with raw data.

This simulation implementation has considered many automatic recovery related aspects in real data center, and is carefully designed so that a reasonable test can be conducted.

Chapter 5 Experimental Analysis

In this chapter, we present how the test is conducted. The evaluation criteria is first clarified, the cost to run-time ratio is defined, the idea of optimal expected cost and the target virtual policy is described. Then the test environment and test setup is elaborated. Finally the test result is presented and analyzed.

5.1 Evaluation Criteria

In order to fairly compare different policy implementations and clearly know which one is better, we defined a simple cost to run-time ratio evaluation criteria. We further studied whether there is a way to always make the best recovery action selection, namely to achieve the best cost to run-time ratio. An optimal expected cost model is defined and associated policy is implemented. Comparing to the best result produced from the optimal policy with the evaluation criteria, we can easily know the performance of each policy.

5.1.1 Cost to Run-time Ratio

Our target is to find a policy with minimal total cost. When comparing different policies, they must be running under the same condition. Obviously, the longer a policy is running, the more cost will be generated. So our comparison is based on the total cost Ct under same total run time RTt, we name it ratio R in our research.

$$R = \frac{Ct}{RTt}$$

$$R = \frac{Ct}{RTt} = \frac{\sum_{i=1}^{n} C_{Si}}{\sum_{i=1}^{n} RT_{Si}}$$

The total cost is composed of C_{Si} the cost from each server Si, and total run time is composed of RT_{Si} the run time from each server Si. In our experiment, we assume that all (n) servers are kept in operation. They are either in the normal servicing, or in the trouble under fixing or trying in probation. We also assume that all servers start from the same time and end at the same time for our experiment, so each server has the same experiment run time RTe. As described in our recovery cost modelling, the cost from each server is composed of the recovery execution downtime and the partial servicing by failed recovery. Both costs are actually counted on each recovery action accumulated cost AC_{Ri} , thus the total cost can be represented by the total accumulated cost from all (m) recovery actions. We can then simplify the data collection and calculation of the ratio R:

$$R = \frac{\sum_{i=1}^{m} AC_{Ri}}{n * RTe}$$

The minimal ratio R represents the minimal total cost under the same run time condition. It's easily to find the best policy by simply selecting the policy with lowest ratio R.

5.1.2 Optimal Expected Cost and Target Policy P0

The initial idea for evaluation was to see how much improvement our policy made comparing to the simple heuristic autopilot base policy. The issue is that we can only know relatively how good or bad the policies are. Even if the result would show which policy is good and which one is bad, and how far they are from each other, we still do not know whether their difference is significant and how far they are from becoming perfect.

In order to find out the absolute position of each policy's total cost result, we need to find either the ultimate worst result or the ultimate best result. The ultimate worst result is hard to answer, the ultimate best result is possible however. Since we make the simulator, we can generate and know all the detailed data behind the scene, such as what is the fault behind each error, what are recovery action properties etc. Thus we can make the best decision for which recovery action should be chosen to have the minimal total recovery cost, which is the best total cost under the current running situation. We call this cost – Optimal Expected Cost.

The calculation of the optimal expected cost among all recovery actions are based on:

- The fault F behind the error
- The recovery action effect probability to above fault
- The unit cost of each recovery action

We do have these data, and they are all generated to form the simulation environment hence the most accurate data in that environment, not the observed data from some policies. So there are no observation errors, no guess work. Thus the best calculation and decision we can make:

$$\underset{Rx \in A}{OEC} = \min_{Rx \in A} \frac{UC_{Rx}}{P_{RxToF}}$$

Based on this formula, we implement policy P0 – our target policy. Because it can always make the best decision, it will produce the best ratio R. We can compare ratio R produced from other policies with policy P0 and improve them continuously to approach perfect. This policy is a virtual policy, as it is based on knowledge of exact fault behind error, exact recovery action effect probabilities to the fault. No policy in real world could know these information. This policy can only exist in a simulator. This is the other advantage of our simulator.

5.2 Test setup and running environment

We setup our experiment on a virtual machine and carefully set the values to the simulation parameters to mimic the real data center characteristics. Then we conducted test in different configurations of simulated servers and total errors.

5.2.1 Running Environment

The physical experiment environment is set on a modern PC with quad core CPU and 4 GB memory. This PC is shared by other tasks. To make the experiment more stable and consistent, we create virtual machine on this PC and run our test inside the virtual machine. The virtual machine is allocated with 1 dedicated processor and 1GB memory and is running in Linux operating system. All simulation code is written in Java and running in a JVM.

The logical environment which is the simulated data center is configured as: total 50 different faults in terms of 50 fault IDs; maximum health time is set to 7200000 seconds which is 2000 hours or 83.33 days on each server; probation length is set to 3600 seconds or 1 hour; number of recovery action is set to 7; minimum recovery action base execution time is set to 5 seconds; maximum recovery action base execution time is set to 36000 seconds in terms of 10 hours; the final recovery action base execution time is set to 360000 seconds in terms of 100 hours, run-time random recovery action execution time floating range is set to within +/-50% of base recovery action execution time.

5.2.2 Test Setup

The test was conducted under different number of simulated servers and total errors, the configurations with different combinations are: 1 server, 100 errors; 1 server, 1000 errors; 1 server, 10000 errors; 10 servers, 100 errors; 10 servers, 1000 errors; 10 servers, 10000 errors; 100 servers, 1000 errors; 100 servers, 10000 errors; 1000 servers, 10000 errors. In order to get statistical meaningful result, the Test Main program is set to have 100 test runs for each configuration.

For each test run, the Test Main creates a new Simulation Controller. Then it generates seeds for all pseudo-random number generators and generates three system static data tables - fault distribution table, recovery action effect table, recovery action base execution time table. After that, there are five sections following, one for each policy. Under each policy section, the simulator along with system state are first initialized. The initialized system state object then is passed into a new created recovery action selector which implements a specified policy. The simulator starts running with that recovery action selector. Finally the simulator reports the result data – total cost to total runtime ratio for that policy. So for one test run, simulation is done five times - one for each policy, and all policies are using exactly the same pseudo-random number generator seeds set and the static data tables. Each policy is running under the same new initialized system state. In other words, each policy is running under the same condition in one test run, thus the result comparison is fair and meaningful. Under one test run, the ratio R of each policy is collected, and the improvement percentage from policy P1, P2, P3, P4 to policy P0 are also collected respectively.

Each test run has new random number generator seeds and new static table data. After 100 times test runs, the statistic result is calculated and reported. The ratio R mean and standard deviation for each policy are calculated. The improvement percentage mean and standard deviation for each pair of policy P1-P4 to policy P0 are also calculated.

5.3 Results

We illustrate the detailed result from one configuration, and summarize results from all configurations. Although our calculation considered both population standard deviation and sample standard deviation [39], the difference is negligible due to the enough test run of 100 times. For clarity, the results only show the population standard deviation.

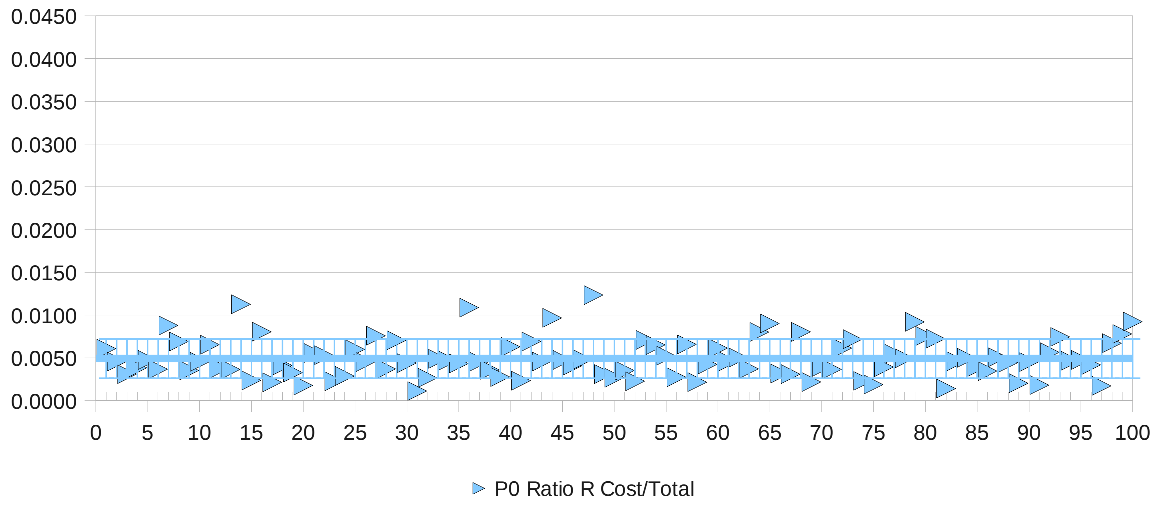


Figure 5.1: Policy P0 Ratio R Detail Result

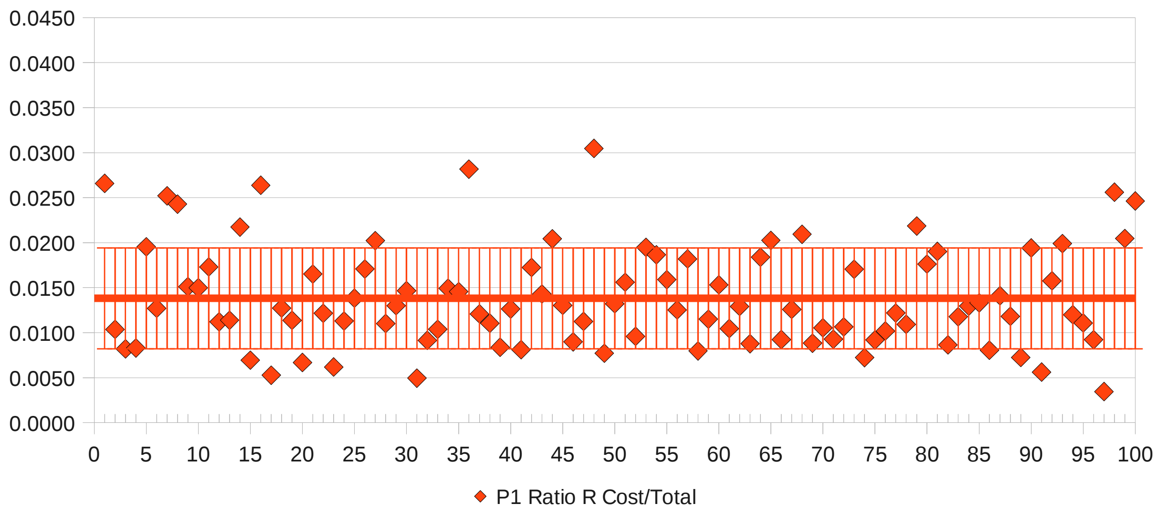


Figure 5.2: Policy P1 Ratio R Detail Result

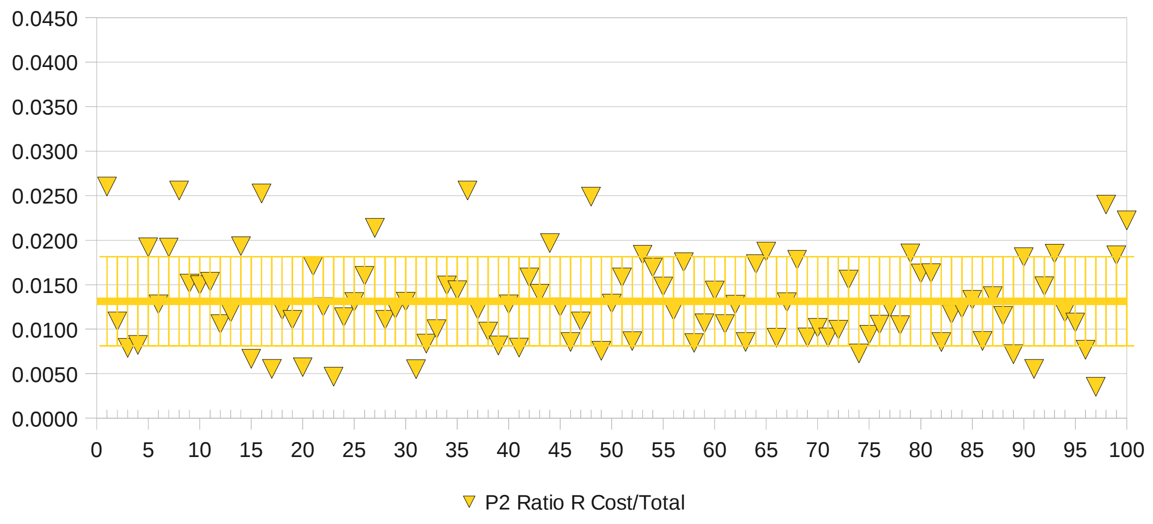


Figure 5.3: Policy P2 Ratio R Detail Result

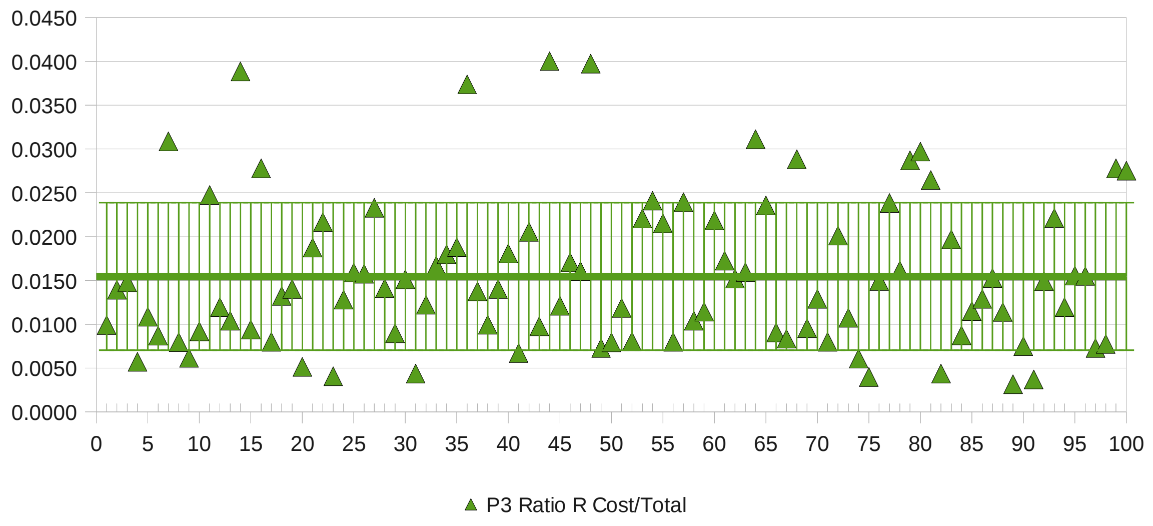


Figure 5.4: Policy P3 Ratio R Detail Result

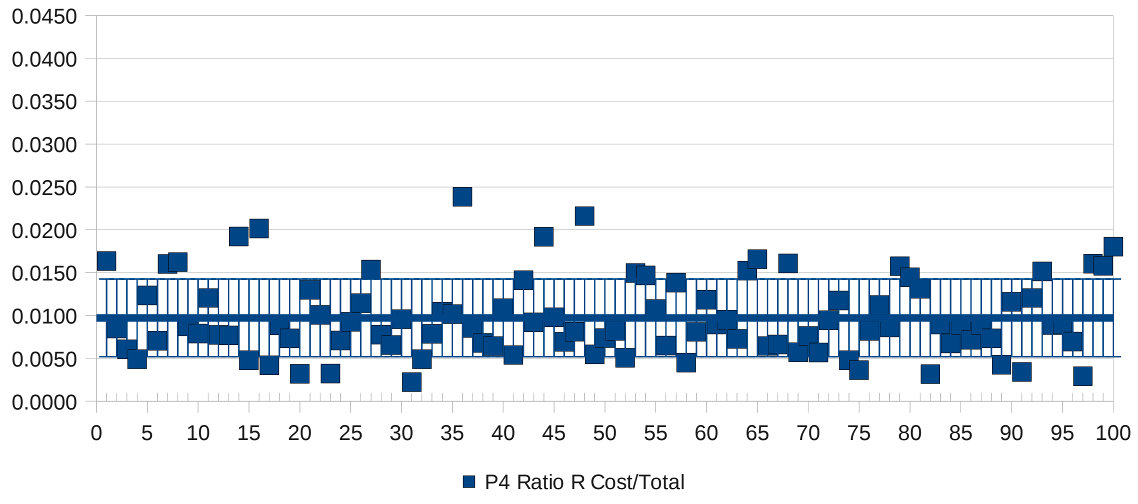


Figure 5.5: Policy P4 Ratio R Detail Result

5.3.1 Detailed Result for 10 servers with 10000 errors

The following diagrams [Figure 5.1, 5.2, 5.3, 5.4, 5.5] show the total cost to total runtime ratio results from the setup of 10 servers and 10000 errors. Summarized in Table 5.1, other than the virtual perfect policy P0, the mean value of policy P4 ratio R is the lowest among other policies, the standard deviation is also the lowest among other policies. The mean value of policy P4 improvement over policy P0 is the best -100.03% whereas the policy P1 improvement over policy P0 is -197.76%, the improvement standard deviation is also the lowest among all other comparable policies. This statistic result clearly demonstrates the lowest total cost advantage of policy P4.

To be more convincing, we not only look at the statistic result but also observe the detailed result from each test run. This will tell us if a policy always has the advantage under the same condition. The Figure 5.6 shows the policy P4 comparing to policy P1, where the result of policy P4 is in the lower line. Under each test run – same condition (same X axis value), the policy P4 in almost all cases has the lower total cost to total runtime ratio (Y axis value) comparing to policy P1. This detailed comparison further convinces the advantage of policy P4.

	Policy P0	**Policy P1**	**Policy P2**	**Policy P3**	**Policy P4**
Mean of ratio R (%)	0.49%	1.38%	1.31%	1.55%	0.97%
Standard deviation of ratio R (%)	0.23%	0.56%	0.50%	0.84%	0.45%
Mean of ratio R Improvement to policy P0 (%)	N/A	-197.76%	-187.25%	-216.09%	-100.04%
Standard deviation of ratio R Improvement to policy P0 (%)	N/A	77.98%	82.04%	86.56%	30.65%

Table 5.1: Ratio R Result Summary of 10 Servers With 10000 Errors

5.3.2 Summarized Results from all configurations

After all configurations of the server and error combinations have been tested, we have found there are some combinations in which the ratio R of policy P4 is worse than policy P1. By analyzing all results in those cases, policy P4 are worse than policy P1 only when there are less errors per server. The Figure 5.7 and 5.8 show the comparison and trends. In the 10 servers setup, when total errors is 10, the total cost to total runtime ratio mean value of policy P4 is very high at 2.50%, and dropping significantly to 1.33% after the total errors reach 100, and further down to 1.10% after the total errors reach 1000, then slowly down to 0.97% when the total errors reach 10000. The policy P1 however is in different trend. Initially, it has

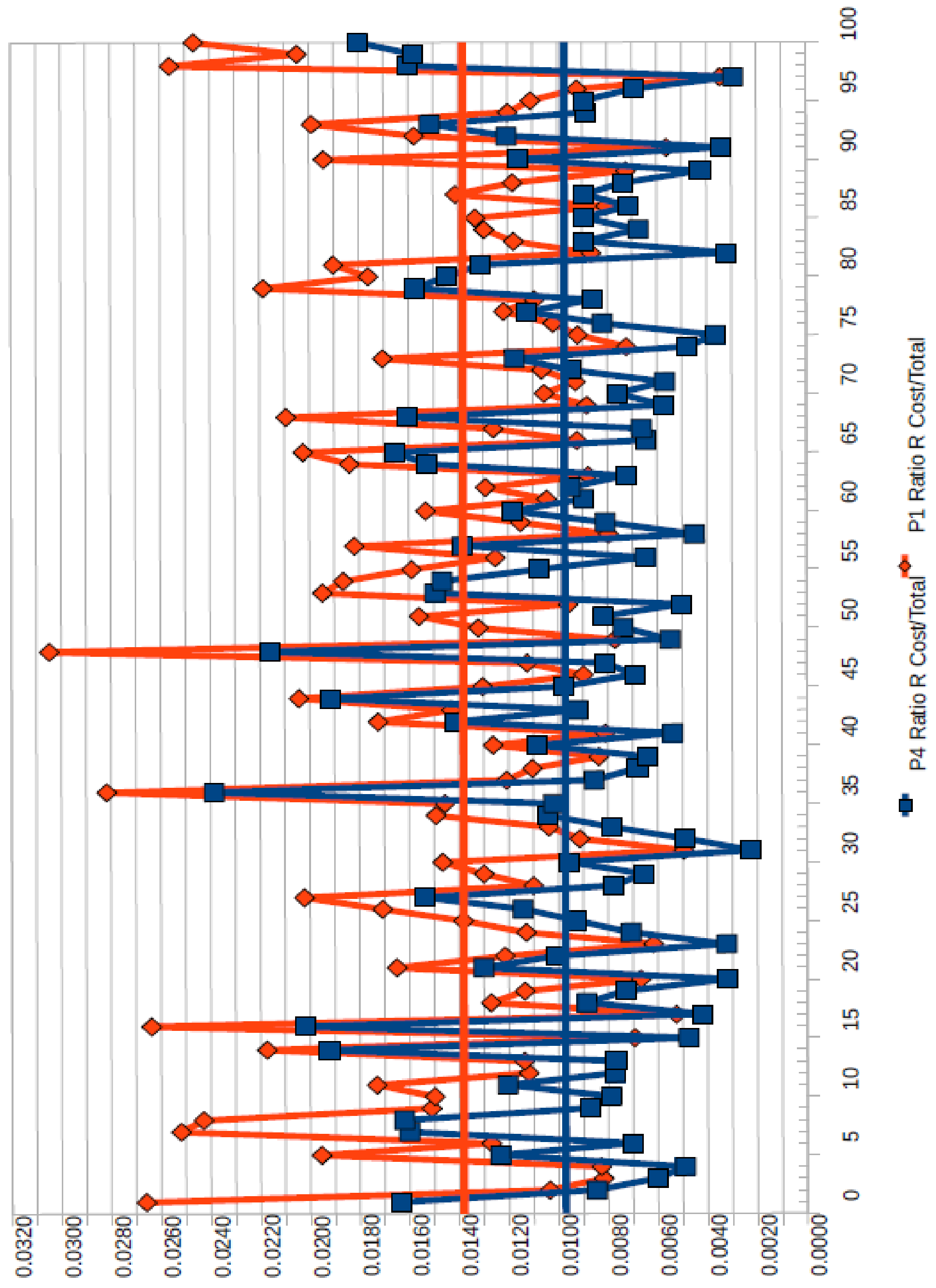


Figure 5.6: Policy P4 to P1 Ratio R Detail Comparison

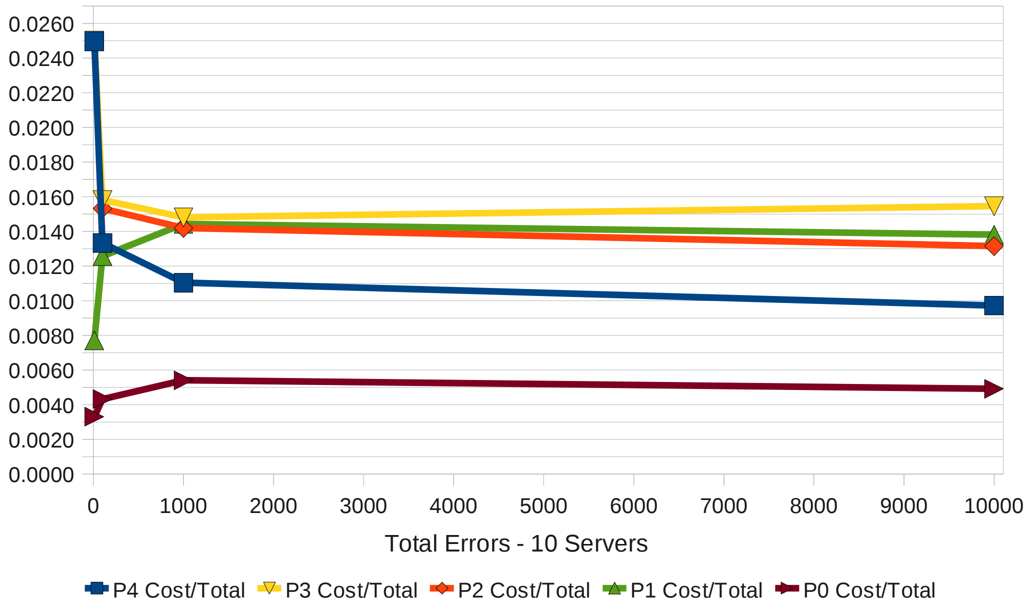


Figure 5.7: 10 Servers Summarized Result

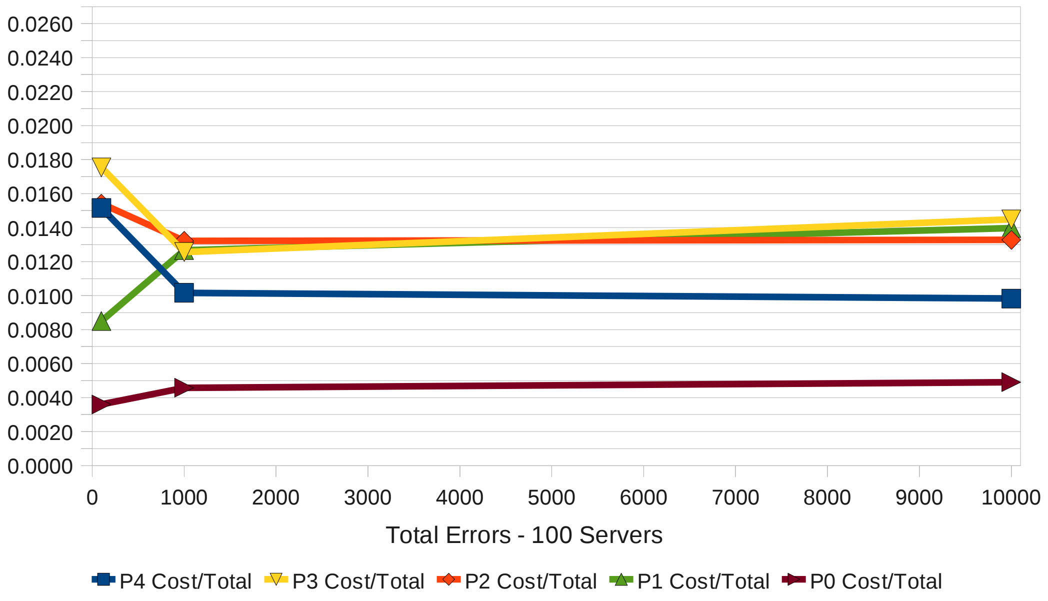


Figure 5.8: 100 Servers Summarized Result

quite low total cost to total runtime ratio mean value at 0.77% when it has just total 10 errors, and significantly rises to 1.26% when it just reaches total 100 errors, then further rises to 1.44% when it encounters total 1000 errors, and stays around that ratio even it encounters total 10000 errors. This similar trend can also be observed from 100 servers setup.

Chapter 6 Policy Improvements

In this chapter, different policy improvements and observations are described. The learning curve observed in our initial experiment results are illustrated and the solution to setup initial value to shorten the learning process is proposed and the improvement is verified in further test. A new observation regarding Attempts per Success Fix is discussed and a test is conducted. Our best policy P4 still shows a good number on this ratio. Finally, we studied the policy P4 multiple attempts coefficient, the results from different coefficient values are analyzed, the best value is suggested.

6.1 Inherent learning process and improvement

As found from the initial experiments, when those servers encounter less errors the total cost to total runtime ratio R of the policy P4 is not better than the ratio R of policy P1. As errors increase to certain amount, the ratio R of policy P4 becomes better and better and exceeds the ratio R of policy P1. This is deemed a learning curve of policy P4, as it needs to learn recovery action unit cost and success probability from recovery attempts to errors, and the selected recovery action output is based on what it learned from those data. The more data it learned the better understanding and selection output it could have. It is observed about 10 errors per server has to be learned. Although the learning process is not that long, any improvement to shorten the learning is still encouraged. That can leverage the benefit from policy P4 as soon as possible and further improve the overall system.

The essence of the idea to shorten the learning is to borrow the knowledge from somewhere instead of learning from the very beginning by itself. As the recovery action unit cost and success probability data are the knowledge the policy has to learn and use, those data can be borrowed from previous running experience – for example from the 1000 errors per server testing run or from data center historical reporting data in real data center situation.

In order to get those statistic recovery action knowledge, our original simulation environment has to be modified. There is a problem in original implementation. For an example of six recovery actions, the recovery action RA1 may have the shortest recovery execution time in one test run but have the longest in another test run, which actually means a different recovery action in real world. The statistic result for recovery action RA1 from different test runs will not make any sense. Hence the recovery action base execution time generator was modified to let recovery action RA1 always have the shortest base recovery execution time and RA6 always have the longest one, and let other recovery actions be fixed in proper order. Therefore, when learning from 100 test runs, the specific recovery action is comparable among different test runs.

Mean of ratio R improvement to Policy P0	**10 servers, 10 errors**	**10 servers, 100 errors**	**10 servers, 1000 errors**	**10 servers, 10000 errors**
Policy P1	-163.24%	-220.03%	-184.79%	-197.76%
Policy P4-Before	-1485.92%	-242.30%	-108.34%	-100.04%
Policy P4-After	-97.95%	-107.66%	-114.34%	-111.66%

Table 6.1: 10 Servers Learning Comparison

Mean of ratio R improvement to Policy P0	**100 servers, 100 errors**	**100 servers, 1000 errors**	**100 servers, 10000 errors**
Policy P1	-155.27%	-194.53%	-203.12%
Policy P4-Before	-399.43%	-131.64%	-99.98%
Policy P4-After	-96.29%	-117.65%	-115.58%

Table 6.2: 100 Servers Learning Comparison

After modification, the test of 10 servers and total 10000 errors was conducted. Then the recovery action RA1 to RAn was learned and their unit cost, accumulated cost, total attempts, success attempts and success probability were summarized from those 100 test runs, their statistic means were collected and input to policy P4 as initial recovery action unit costs and initial success probabilities. After these knowledge data gets input to policy P4, another set of tests was conducted.

The result comparison (Table 6.1 and 6.2) clearly demonstrates the improvement on learning curve. In the 10 servers setup testing, without the learning knowledge, the ratio R of policy P4 improvement to policy P0 in less errors case is clearly worse than policy P1's result, until 1000 errors encountered the policy P4 shows its advantage. After putting the knowledge, the difference is significant. From the initial 10 errors, the result shows the ratio R of policy P4 improvement to policy P0 changed from -1485.92% to -97.95%. Then, for 100 errors, the policy P4 result also changes from -242.3% to -107.66%. Similar results are also shown in the 100 servers setup. In both setup, the policy P4 shows better result than policy P0 from 10 errors. The results clearly demonstrate that our approach significantly improves the policy P4 in the early learning stage, the advantage of total cost to total runtime ratio of policy P4 immediately shows from the beginning.

6.2 Observation of Attempts per Success Fix

While the policy P4 is demonstrated as the best from the total cost perspective. The question is raised about if there is any other impacts. We know if the final recovery action is selected, any error can always be fixed successfully in terms of just one attempt, however that will introduce the highest cost. While a policy maintains a low total cost, can it also have a low average attempts per success fix (we name it R2)?

How to calculate this ratio? For example, an error happens, and an recovery action is applied however failed, another error happens in probation, another recovery action is applied and failed again, once more an error happens in probation, third recovery action is applied and does a successful fix. After a long period a new error happens, and a recovery action is applied and succeed. As the above example there are two original errors, the other two happen because of failed recovery action attempts. Hence there are total four errors and total four recovery action attempts, and two successful fixes. So that example has average 2 attempts per success fix – 4 attempts divided by 2 successful fix. The total recovery action attempts (TA) is simply equals total errors (TE), because there is always a recovery action selected for an error. The total success fix number (TF) can be inferred from total original errors which is hard to track and distinguish, or can count from how many times a server changes state to Health which simply and clearly means a success fix happened. The ratio R2 should be counted from all (n) servers, while each server can collect the A_{Si} recovery action attempts (A) on that server (Si) and collect the F_{Si} success fix number (F) on that server (Si), see formula 6.1A and 6.1B.

$$R2=\frac{TA}{TF}=\frac{\sum_{i=1}^{n} A_{Si}}{\sum_{i=1}^{n} F_{Si}} \tag{6.1 A}$$

$$R2=\frac{TE}{TF}=\frac{TE}{\sum_{i=1}^{n} F_{Si}} \tag{6.1 B}$$

We further modified the simulator to capture this ratio R2, and did test from the setup of 10 servers and 10000 errors. The detailed results are showed in Figure 6.1, 6.2, 6.3, 6.4, 6.5. The summarized result in Table 6.3 shows the mean value of attempts per success fix ratio of policy P4 is similar as policy P0, and is close to but slightly worse than policy P1, the ratio R2 standard deviation of policy P4 is also close and slightly worse than policy P1. The ratio R2 of policy P2 is also close to policy P1, however shows small improvement. The ratio R2 of policy P3 is significant worse than policy P1, and is the worst among all policies. This statistic result shows: while policy P4 maintains the lowest total cost advantage, it also maintains a comparable attempts per success fix ratio and does not have significant impact to that.

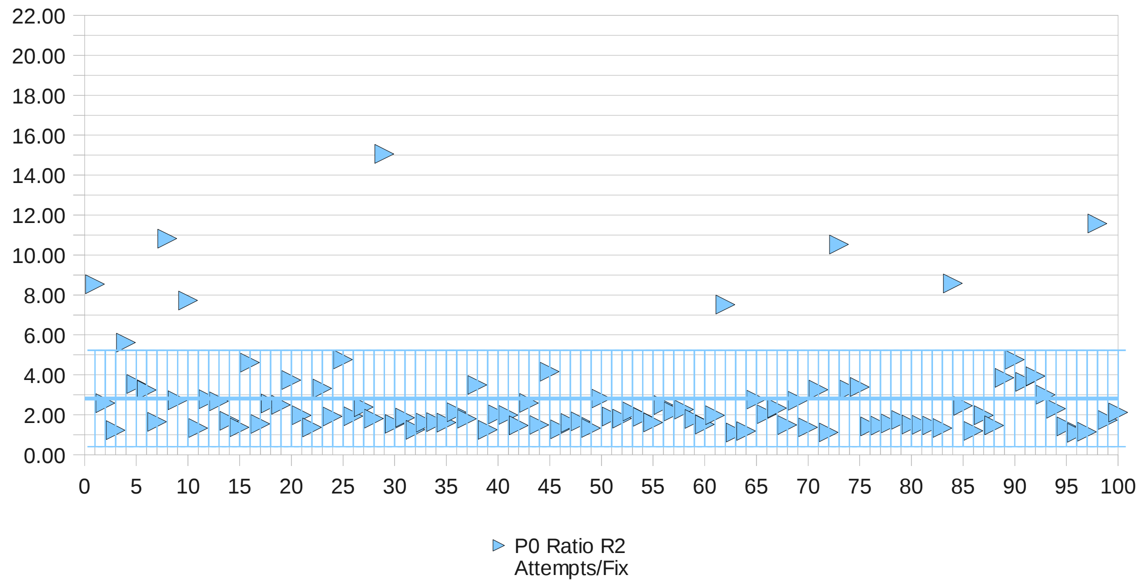


Figure 6.1: Policy P0 Ratio R2 Detail Result

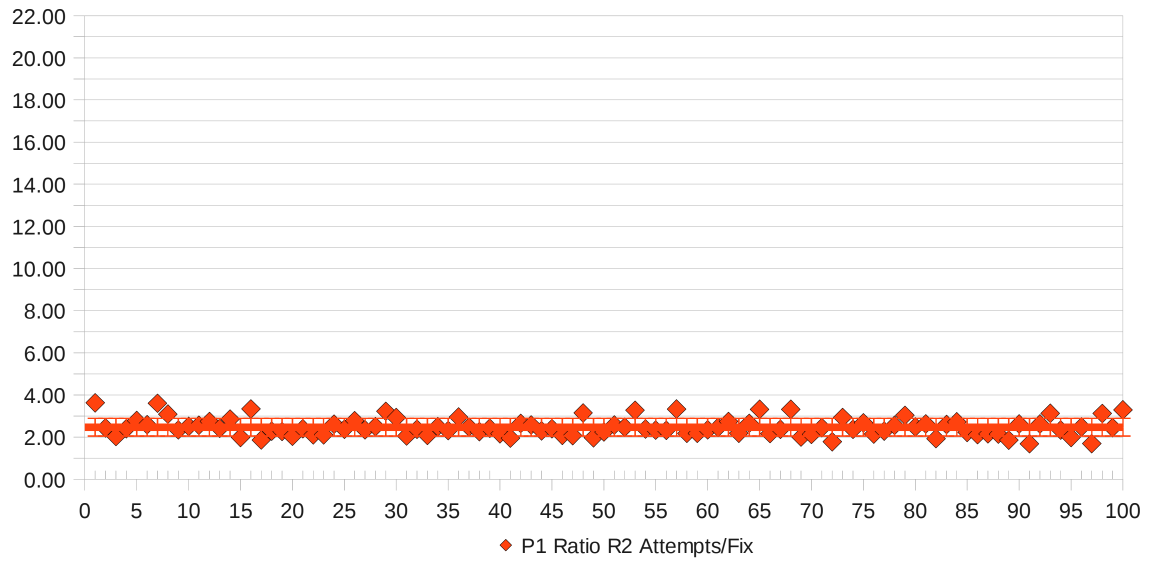


Figure 6.2: Policy P1 Ration R2 Detail Result

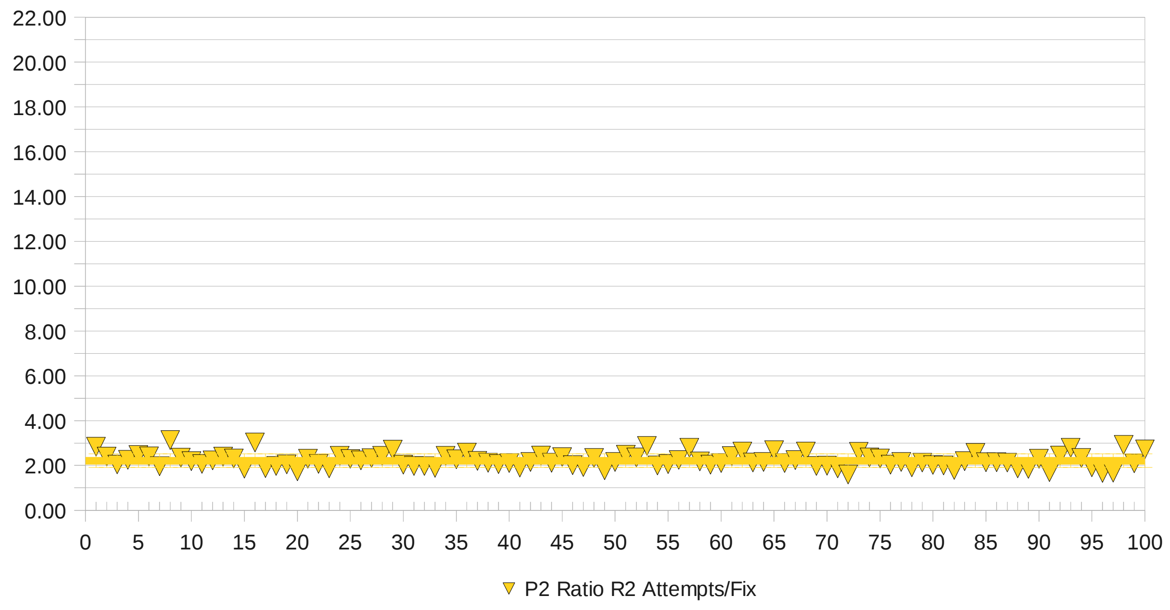


Figure 6.3: Policy P2 Ratio R2 Detail Result

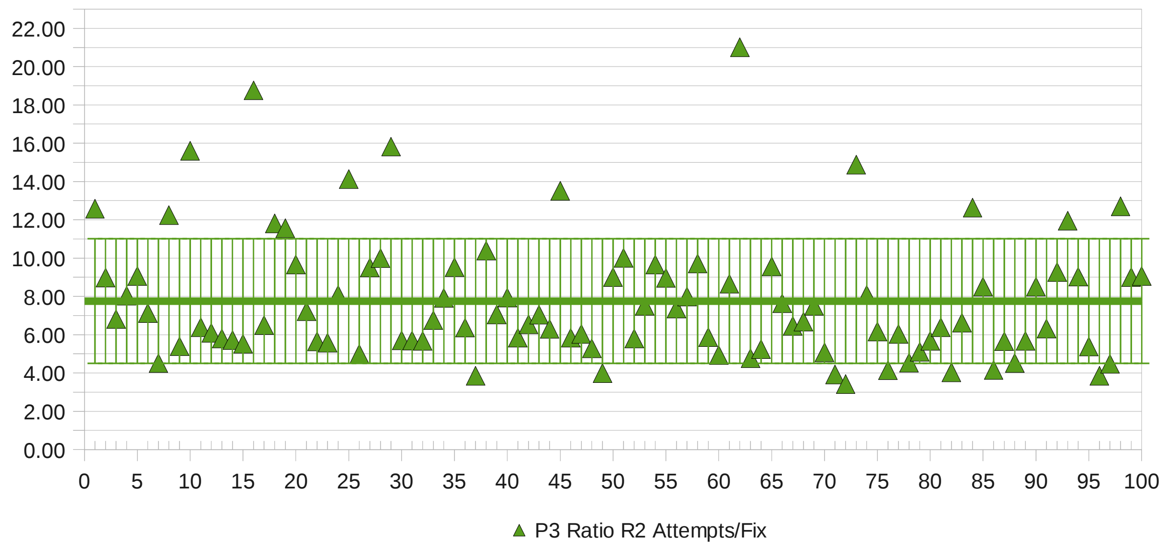


Figure 6.4: Policy P3 Ratio R2 Detail Result

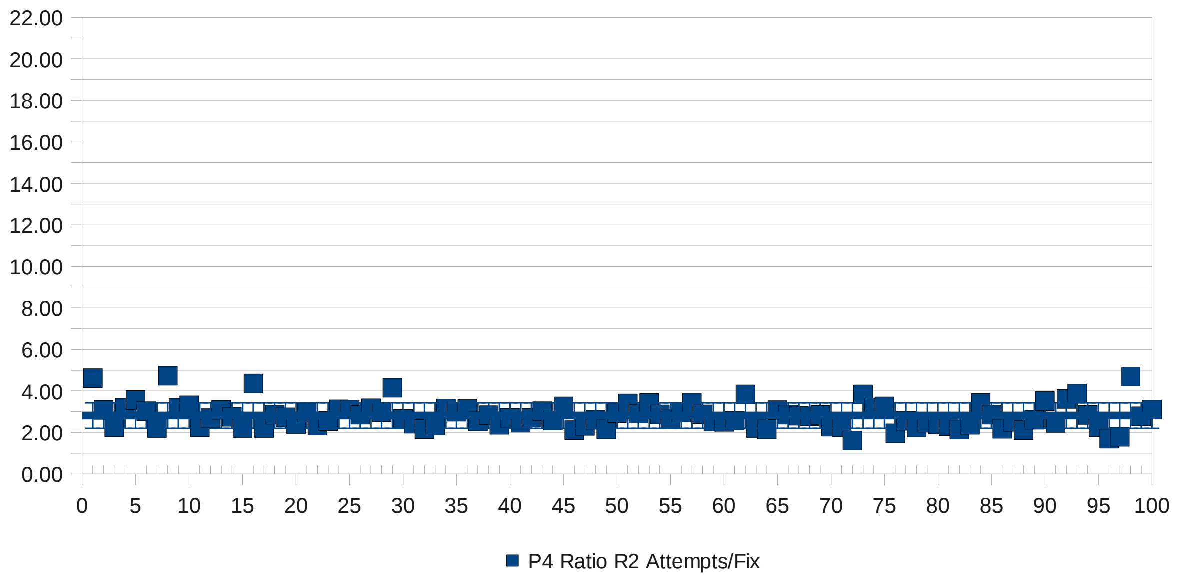


Figure 6.5: Policy P4 Ratio R2 Detail Result

	Policy P0	**Policy P1**	**Policy P2**	**Policy P3**	**Policy P4**
Mean of ratio R2	2.82	2.47	2.21	7.76	2.81
Standard deviation of ratio R2	2.41	0.42	0.31	3.25	0.61
Mean of ratio R2 Improvement to policy P0 (%)	N/A	-19.67%	-6.34%	-237.22%	-29.34%
Standard deviation of ratio R2 Improvement to policy P0 (%)	N/A	50.32%	41.12%	114.37%	42.05%

Table 6.3: Ratio R2 Result Summary of 10 Servers With 10000 Errors

6.3 Policy P4 Multiple Attempts Coefficient Study

As mentioned in the policy implementation, the policy P3 has the basic implementation of our total cost based solution. It never considers the difference of second and multiple recovery attempts with the same recovery action. Policy P2 considers this by eliminating the second and multiple recovery attempts with the same recovery action. And policy P4 considers this by reducing the success probability of the second and multiple recovery attempts with the same recovery action. The policy P4 reduces the success probability by multiplying additional multiple (n) attempts coefficient 0.5^n to the original recovery action success probability. Through further analysis, the policy P2 and policy P3 are actually the extreme cases of policy P4. Policy P3 is actually the same as setting the policy P4 coefficient to 1^n, so there is no difference of second or multiple recovery attempts, their coefficients are all equal to 1 namely their success probability always equals to the original recovery action success probability. Policy P2 is actually the same as setting the policy P4 coefficient to 0^n, so there is no chance for the second or more recovery attempts, their coefficients are all equal to 0 namely 0 success probability. We modified the policy P4 coefficient to 1 and 0 respectively and did tests for 10 servers and total 1000 errors. As expected it shows exactly same result to policy P2 and P3 respectively. See Table 6.4, 6.5 and 6.6. The verified results also demonstrate our implementation did give fair chance to each policy and our implementation was built correctly from another angle.

Coefficient=0.5^n	**Policy P4**	**Policy P3**	**Policy P2**
Mean of Cost/Total	1.10%	1.48%	1.42%
Standard deviation of Cost/Total	0.55%	0.70%	0.62%
Mean of Attempts/Fix	2.83	5.66	2.28
Standard deviation of Attempts/Fix	0.67	2.09	0.37

Table 6.4: Policy P4 Coefficient 0.5 Comparison

Coefficient=0.0^n	**Policy P4**	**Policy P3**	**Policy P2**
Mean of Cost/Total	1.37%	1.36%	1.37%
Standard deviation of Cost/Total	0.60%	0.65%	0.60%
Mean of Attempts/Fix	2.27	5.79	2.27
Standard deviation of Attempts/Fix	0.43	2.16	0.43

Table 6.5: Policy P4 Coefficient 0.0 Comparison

Coefficient=1.0^n	**Policy P4**	**Policy P3**	**Policy P2**
Mean of Cost/Total	1.37%	1.37%	1.41%
Standard deviation of Cost/Total	0.63%	0.63%	0.65%
Mean of Attempts/Fix	5.81	5.81	2.30
Standard deviation of Attempts/Fix	1.86	1.86	0.42

Table 6.6: Policy P4 Coefficient 1.0 Comparison

Based on our simulation platform, we further studied the policy P4 multiple attempts coefficient to find the impacts to both total cost to total runtime ratio R and attempts per success fix ratio R2, and also to find the best coefficient value. To avoid the variations among different tests, the simulation platform is modified to run and collect the results from different coefficient setups in each test run. The studied coefficients are from 0.0 to 1.0 in 0.1 stepping. A test was conducted in 10 servers and total 10000 errors configuration. Both ratios are collected and presented to show their trends respectively.

The summarized results for ratio R are shown in Figure 6.6. When coefficient value equals 0, the policy P4 gets the high total cost to total runtime ratio R as 0.013142. The ratio R decreases gradually and reach the lowest ratio R 0.00938 when coefficient value equals

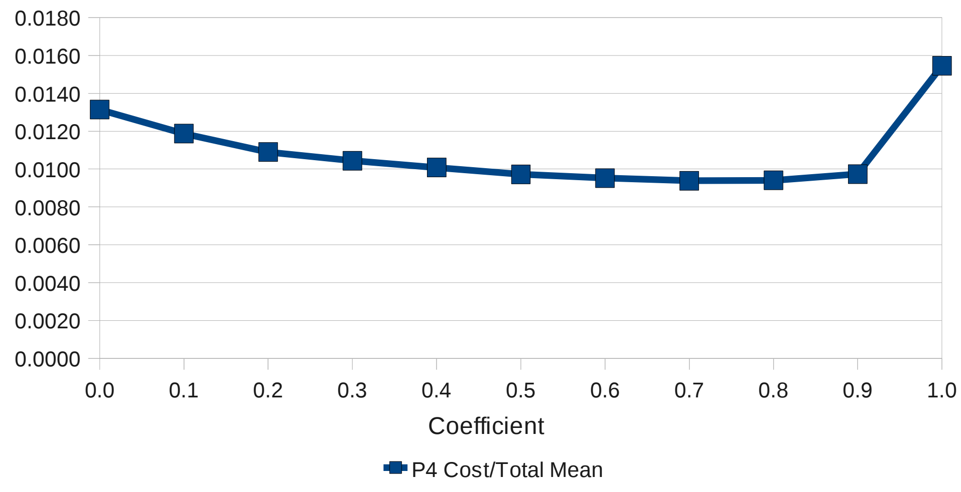


Figure 6.6: Policy P4 Ratio R Coefficient Result

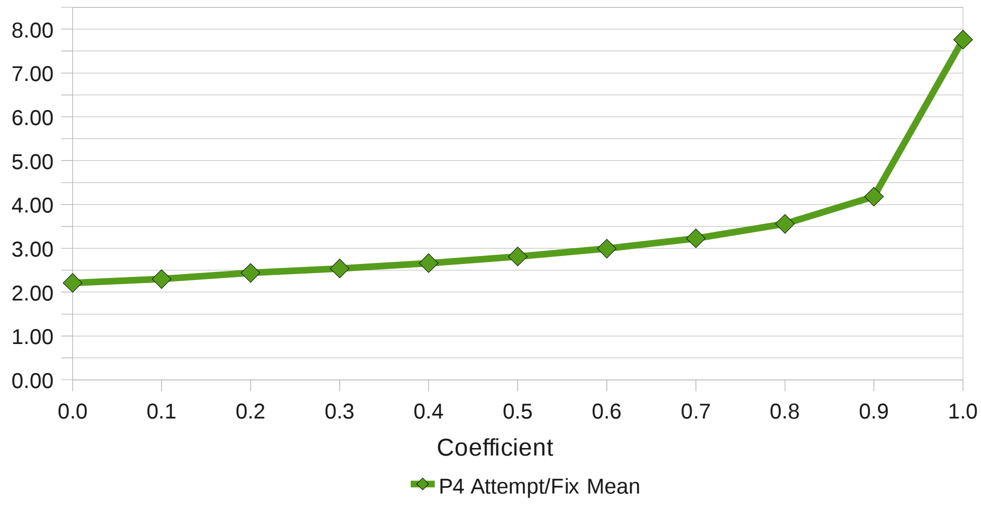


Figure 6.7: Policy P4 Ratio R2 Coefficient Result

0.7, then the ratio R slightly increases. After the coefficient value passes 0.9 the ratio significantly increases and reaches the highest ratio R 0.0154528 when coefficient value equals 1.

The summarized results for ratio R2 are shown in Figure 6.7. When coefficient value equals 0, the policy P4 gets the lowest attempts per success fix ratio R2 as 2.212857. The ratio R2 gradually increases to 4.1801468 until coefficient value increases to 0.9. After that point the ratio significantly increases and reaches the highest ratio 7.760022 when coefficient value equals 1.

If the lowest total cost to total runtime is the only target, the coefficient value of 0.7 is the best to choose. The coefficient value of 0.0 is the best for reaching minimum attempts per success fix. And the coefficient value around 0.5 is the reasonable compromise to both.

Chapter 7 Conclusion and Future Work

This thesis studies the automatic recovery policy problem - how to select a proper recovery action without knowing fault information. The target is to achieve minimal total cost. We also study the cost from the recovery action execution down time and the partial service caused by failed recovery. The recovery action success probability is studied as well. We propose an estimated total cost ETC model to select the recovery action. This generic cost based model utilizes the dynamic updated statistic data (recovery action unit cost and success probability), and selects the recovery action with minimal estimated total cost from a set of recovery action candidates. Based on this model, three policies are implemented. These policies make different adjustments to the recovery action success probability based on the recovery action failed attempts. Our generic cost based policy implementation is self adaptive to the system, it can bootstrap reasonably and adjust itself automatically.

In addition, this thesis analyzes the fault distribution and recovery action to fault effect probability, and the recovery action execution time variation. Based on these analyses, we implement a discrete event simulator to properly mimic the data center automatic recovery operation environment.

As our simulation implementation, we are able to define an optimal expected cost and implement a virtual perfect policy based on that calculation. Our experiment results eventually show that we have achieved our goal to reduce the total recovery cost. Comparing to a similar research with heuristic model, our policy to the virtual perfect policy improvement is only -100% whereas the heuristic policy is -198%, the result shows big improvement made by our policy.

We further provide improvement solution to shorten the learning curve in our model by inputting reasonable knowledge in advance. We also study Attempts per Success Fix ratio, and our policy shows minimal impacts on this ratio. Finally we carefully study the multiple attempts coefficient and give a recommended value. The optimized coefficient makes the total cost to run-time ratio even better.

7.1 Future Work

Our model is a recursive model, and we have simplified the model not to include the repeating recovery actions and deal with that issue in policy by reducing repeating recovery action success probability. However there is still deep recursive calculation, we would like to find better simplified calculation algorithm. Also we simply select the power formula to reduce success probability for multiple attempts of failed recovery actions. We would like to study other alternatives for improvement.

We have also mentioned different cost scenarios like human cost, business loss, contract penalty and other monetary or non monetary based cost etc. But we have not made further researches on this area. The potential solution could be to build a cost formulation according to real environment and feed the translated cost back to our model. As mentioned in ROC, not all downtime are equal, and they are not always linear relationship [12]. By plugging in the organization dependent cost formulation, the weight on downtime could be more realistic and the recovery action selection will be more suitable for the organization.

Another improvement that could be made to the policy is to extend it to take more dimensions or parameters such as: different failure events, computer configuration signature – OS version, HW configuration etc. In real data center, machines may be updated every year and both operating systems and application softwares are kept updating. These changes inhere with different statistic characteristics. In order to capture the impacts from those dimensions in the real environment, the policy could be extended. The generic cost based model could be still kept the same. The recovery action unit cost and success probability recording and calculation could be modified. They could be extended to also record the unit cost and success probability under other dimensions (e.g. hardware configuration signature and software versions) on top of the existing overall average unit cost and success probability. Both recovery action unit cost and success probability calculation can be modified to consider the more specific recovery action unit cost and success probability values versus the generic overall values. The calculation could: 1) always utilize the specific values; 2) use percentage from overall generic value and percentage from specific values; 3) dynamically switch between above two approaches - by considering the early phase of a change, there is no enough sampling to conclude a mature value; then we can take percentage from overall generic value and percentage from specific values when the sampling is less than certain amount (e.g. 1000); after enough sampling is collected, we can use the more accurate specific values. The dimension could be easily extended as the recovery action could be extended, and the specific recovery action unit cost and success probability values are automatically collected and updated. Consequently the policy also automatically adopts the changing values, and the system is automatically optimized from the added dimensions.

Finally, we would like to run our policy in a real data center for a long period if such arrangement is allowed. After enough simulation, we would like to verify and improve our model in the real world.

References

[1] Amina Khalid, Mouna Abdul Haye, Malik Jahan Khan, and Shafay Shamail. Survey of Frameworks, Architectures and Techniques in Autonomic Computing. In *IEEE Computer Society*, 220-225, 2009

[2] Luiz André Barroso and Urs Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Morgan & Claypool Publishers*, 2009 .

[3] Moises Goldszmidt, Mihai Budiu, Yue Zhang, and Michael Pechuk. Toward Automatic Policy Refinement in Repair Services for Large Distributed Systems. In SIGOPS Operating System Review, Volume 44, Issue 2, April 2010

[4] Andreas Hanemann, David Schmitz, and Martin Sailer. A Framework for Failure Impact Analysis and Recovery with Respect to Service Level Agreements. In *Proceedings of the IEEE International Conference on Services Computing (SCC 2005),* Orlando, Florida, USA: IEEE, July 2005.

[5] Guy Shani and Christopher Meek. Improving Existing Fault Recovery Policies. In *Advances in Neural Information Processing Systems 22*, pages 1642-1650, 2009

[6] Michael Isard. Autopilot: Automatic Data Center Management. In *Operating Systems Review*, 41:60–67, 2007.

[7] Moises Goldszmidt, Miroslaw Malek, Simin Nadjm-Tehrani, Priya Narasimhan, Felix Salfner, Paul A.S. Ward, and John Wilkes. Wheels within Wheels: Making Fault Management Cost-Effective. In *Dagstuhl Seminar Proceedings 09201 , Combinatorial Scientific Computing* , Dagstuhl, Germany , May 10–15, 2009 .

[8] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. In *IEEE Computer,* 36(1):41–50, January 2003.

[9] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An Architectural Approach to Autonomic Computing. In *Proceedings of the International Conference on Autonomic Computing (ICAC),* 2004

[10] Olga Brukman, Shlomi Dolev, Yinnon Haviv, and Reuven Yagel. Self-Stabilization as a Foundation for Autonomic Computing. In *The Second International Conference on*

Availability, Reliability and Security (ARES), pages 991–998, Vienna, April 2007.

[11] Pushkar Kumar. Autonomic Computing, A Seminar Report. Bachelor of Technology thesis, September 2008.

[12] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kıcıman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, March 15, 2002.

[13] George Candea, Emre Kiciman, Shinichi Kawamoto, and Armando Fox . Autonomous Recovery in Componentized Internet Applications. In *Cluster Computing Journal*, Vol. 9, No. 2, April 2006.

[14] Xiaolin Li, Hui Kang, Patrick Harrington, and Johnson Thomas. Autonomic and Trusted Computing Paradigms. In *ATC 2006, LNCS 4158*, pages 143-152, 2006

[15] Mohammad A. Munawar and Paul A.S. Ward. Leveraging many simple statistical models to adaptively monitor software systems. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2007.

[16] Mohammad A. Munawar, Miao Jiang, Allen George, Thomas Reidemeister, and Paul A. S. Ward. Adaptive monitoring with dynamic differential tracing- based diagnosis. In *Proceedings of the 19th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, 2008.

[17] Mohammad A. Munawar, Paul A.S. Ward, and Kevin Quan. Interaction analysis of heterogeneous monitoring data for autonomic problem determination. In I*EEE International Symposium on Ubisafe Computing. IEEE Computer Society Press, 2007*.

[18] Dorron Levy and Ram Chillarege. Early Warning of Failures through Alarm Analysis - A Case Study in Telecom Voice Mail Systems. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, Washington, DC, USA, 2003. IEEE Computer Society.

[19] Adam Oliner and Jon Stearley. What Supercomputers Say: A Study of Five System Logs. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on*

Dependable Systems and Networks, pages 575–584, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Mohammad A. Munawar. Adaptive Monitoring of Complex Software Systems using Management Metrics. PhD thesis, Electrical and Computer Engineering Division, University of Waterloo, 2009. http://hdl.handle.net/10012/4797

[21] Rajkumar Buyya, Rajiv Ranjan and Rodrigo N. Calheiros. Modeling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities, Keynote Paper. In *Proceedings of the 7th High Performance Computing and Simulation (HPCS 2009) Conference*, Leipzig, Germany (2009)

[22] Ilango Sriram. SPECI, a simulation tool exploring cloud-scale data centres. In *CloudCom 2009*, *LNCS 5931*, pages 381-392, M.G. Jaatun, G. Zhao, and C. Rong (Eds.), Springer-Verlag Berlin, Heidelberg 2009, http://arxiv.org/abs/0910.4568

[23] Ian Foster, Yong Zhao, Ioan Raicu, Shiyong Lu . Cloud Computing and Grid Computing 360-Degree Compared. In *Proceedings IEEE Grid Computing Environments Workshop*, pages 1-10, 2008.

[24] Simon Wardley, Etienne Goyer, and Nick Barcet. Technical White Paper Ubuntu Enterprise Cloud Architecture.
http://www.ubuntu.com/system/files/UbuntuEnterpriseCloudWP-Architecture-20090820.pdf

[25] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-source Cloud Computing System. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2009)*, Shanghai, China, pages124–131, 2009. http://open.eucalyptus.com/documents/ccgrid2009.pdf

[26] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk , Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. Eucalyptus: a technical report on an elastic utility computing architecture linking your programs to useful systems. Technical Report 2008-10, Department of Computer Science, University of California, Santa Barbara, California, USA, 2008.
http://open.eucalyptus.com/documents/nurmi_et_al-eucalyptus_tech_report-august_2008.pdf

[27] Rodrigo N. Calheiros, Rajiv Ranjan, César A. F. De Rose, and Rajkumar Buyya.

CloudSim: A Novel Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services. Technical Report, Grid Computing and Distributed Systems (GRIDS) Laboratory, Department of Computer Science and Software Engineering, The University of Melbourne, 2009.

[28] Kashi Venkatesh Vishwanath, Albert Greenberg, and Daniel A. Reed. Modular data centers: how to design them?. In *Proceedings of the 1st ACM Workshop on LSAP,* 2009.

[29] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in MapReduce setup. In *International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, London, UK, September 2009.

[30] Christian Baun. Elastic Cloud Computing in the Open Cirrus Testbed implemented via Eucalyptus. In *ISGC 2009. Forthcoming 2009 LNCS*, Springer, Heidelberg

[31] David Dyer. Current trends/challenges in datacenter thermal management - a facilities perspective, presentation at ITHERM, San Diego, CA, June 1, 2006.

[32] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[33] Recovery-Oriented Computing Overview. http://roc.cs.berkeley.edu/roc_overview.html

[34] Pete Broadwell, Naveen Sastry, and Jonathan Traupman. FIG: A Prototype Tool for Online Verification of Recovery Mechanisms. In *Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, New York, NY, June 2002.

[35] George Candea , Aaron B. Brown, Armando Fox, and David Patterson. Recovery-Oriented Computing: Building Multitier Dependability . In *IEEE Computer,* 37(11):60–67, 2004.

[36] Mike Y. Chen, Emre Kıcıman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic systems. In *Proceedings 2002 Intl. Conf. on Dependable Systems and Networks,* pages 595–604, Washington, DC, June 2002.

[37] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery . In *Proceedings of the 6th Symposium on Operating System Design and Implementation,* December 2004.

[38] Aaron B. Brown and David A. Patterson. To Err is Human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependability (EASY '01),* Goeteborg, Sweden, July 2001.

[39] David Stone and Jon Ellis. Stats Tutorial – Mean, Variance and Standard Deviation. University of Toronto, 2006.
http://www.chem.utoronto.ca/coursenotes/analsci/StatsTutorial/MeasMeanVar.html

[40] Aaron B. Brown and David A. Patterson. Undo for Operators: Building an Undoable E-mail Store . In *Proceedings of the 2003 USENIX Technical Conference*, June 2003.

[41] Aaron Brown. A Recovery-Oriented Approach to Dependable Services: Repairing Past Errors with System-Wide Undo. PhD thesis, Computer Science Division, University of California, Berkeley, 2003.

[42] Aaron B. Brown and David A. Patterson. Rewind, Repair, Replay: Three R's to Dependability. In *Proceedings 10th ACM SIGOPS European Workshop*. St. Emilion, France, 2002.

[43] David Oppenheimer, Aaron Brown, James Beck, Daniel Hettena, Jon Kuroda, Noah Treuhaft, and David A. Patterson. ROC-1: Hardware Support for Recovery-Oriented Computing . In *IEEE Transactions on Computers, vol. 51, no. 2,* February 2002.

[44] Yennun Huang, Chandra Kintala, Nick Kolettis and N. Dudley Funton. Software Rejuvenation: Analysis, Module and Applications. In *Proceeding IEEE Int'l Symposium on Fault Tolerant Computing, IEEE Computer Society Press*, Los Alamitos, CA, 1995, pages 381–390.

[45] Tadashi Dohi, Katerina Goseva-Popstojanova, and Kishor S. Trivedi. Statistical Non-Parametric Algorithms to Estimate the Optimal Software Rejuvenation Schedule. In *Proceeding International Pacific Rim Symposium on Dependable Computing*, pages 77–84, 2000.

[46] Kalyanaraman Vaidyanathan, Richard E. Harper, Steven W. Hunter, and Kishor S. Trivedi. Analysis and Implementation of Software Rejuvenation in Cluster Systems. In *Proceeding ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pages 62– 71, 2001.

[47] Sachin Garg, Yennun Huang, Chandra Kintala, and Kishor S. Trivedi. Time and Load

Based Software Rejuvenation: Policy, Evaluation and Optimality. In *Proceedings of the First Fault- Tolerant Symposium*, Madras, India, December 22–25, 1995.

[48] Andrea Bobbio, Matteo Sereno, and Cosimo Anglano. Fine grained software degradation models for optimal rejuvenation policies . In *Performance Evaluation*, vol. 46, no. 1, pages 45-62, 2001

[49] Theo Hearder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. In *ACM Computing Surveys 15, 4 (December 1983). Reprinted in M. Stonebraker, Readings in Database Sys.*, Morgan-Kaufman, San Mateo, CA, 1988.

[50] Raymond A. Lorie. Physical Integrity in a Large Segmented Database. In *ACM Trans Database Syst 2,* 1 (March 1977), 91-104.

[51] United States Department of Labor, Bureau of Labor Statistics, Employer Costs for Employee Compensation. September 2010. http://www.bls.gov/news.release/ecec.t11.htm

[52] Nancy Gohring. Microsoft to build giant data center in Virginia, August 2010. http://www.infoworld.com/d/the-industry-standard/microsoft-build-giant-data-center-in-virginia-855

[53] Rich Miller. Amazon Building Large Data Center in Oregon, November 2008. http://www.datacenterknowledge.com/archives/2008/11/07/amazon-building-large-data-center-in-oregon/

[54] Matt Stansberry. Data center locations ranked by operating cost. July 2006. http://searchdatacenter.techtarget.com/news/1204203/Data-center-locations-ranked-by-operating-cost

[55] Bianca Schroeder and Garth A. Gibson. Understanding Failures in Petascale Computers. In *Journal of Physics: Conference Series*, 78:012022 (11pp), 2007.

[56] Daniel A. Menascé. Performance and Availability of Internet Data Centers . In *IEEE Internet Computing,* May/June 2004, Vol. 8, No. 3, 94-96

[57] TIA Standard, Telecommunications Infrastructure Standard for Data Centers, TIA-942 . April 2005.

[58] Contingency Planning Research (http://www.contingencyplanningresearch.com), a Division of Eagle Rock Alliance (http://www.eaglerockalliance.com)

[59] Rich Miller. Who Has the Most Web Servers? May 2009.

http://www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers/
[60] Simon Mingay. Look Beyond Google's Plan to Become Carbon Neutral . *Gartner Publisher*, June 2007.
[61] Wikipedia. Cloud computing. http://en.wikipedia.org/wiki/Cloud_computing
[62] Rich Miller. How Many Servers Can One Admin Manage? December 2009.
http://www.datacenterknowledge.com/archives/2009/12/30/how-many-servers-can-one-admin-manage/