THE UNIVERSITY OF
WARWICK

University of Warwick institutional repository: http://go.warwick.ac.uk/wrap

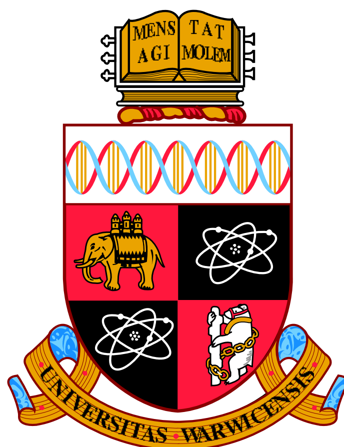
A Thesis Submitted for the Degree of PhD at the University of Warwick

http://go.warwick.ac.uk/wrap/77699

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.



Towards Efficient Error Detection in Large-Scale HPC Systems

by

Nentawe Yusuf Gurumdimma

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

February 2016

Abstract

The need for computer systems to be reliable has increasingly become important as the dependence on their accurate functioning by users increases. The failure of these systems could very costly in terms of time and money. In as much as system's designers try to design fault-free systems, it is practically impossible to have such systems as different factors could affect them. In order to achieve system's reliability, fault tolerance methods are usually deployed; these methods help the system to produce acceptable results even in the presence of faults. Root cause analysis, a dependability method for which the causes of failures are diagnosed for the purpose of correction or prevention of future occurrence is less efficient. It is reactive and would not prevent the first failure from occurring. For this reason, methods with predictive capabilities are preferred; failure prediction methods are employed to predict the potential failures to enable preventive measures to be applied.

Most of the predictive methods have been supervised, requiring accurate knowledge of the system's failures, errors and faults. However, with changing system components and system updates, supervised methods are ineffective. Error detection methods allows error patterns to be detected early to enable preventive methods to be applied. Performing this detection in an unsupervised way could be more effective as changes to systems or updates would less affect such a solution. In this thesis, we introduced an unsupervised approach to detecting error patterns in a system using its data. More specifically, the thesis investigates the use of both event logs and resource utilization data to detect error patterns. It addresses both the spatial and temporal aspects of achieving system dependability. The proposed unsupervised error detection method has been applied on real data from two different production systems. The results are positive; showing average detection F-measure of about 75%.

This thesis is dedicated to the memory of my dear brother, friend and a mentor.

Filibus Istifanus G.

(1970 - 2014)

Acknowledgements

I am indebted to many people who have helped me in one way or the other during my studies at University of Warwick. I am honoured to acknowledge them in this thesis.

First and foremost, I am grateful to Dr. Arshad Jhumka, whose meticulous and relentless guide is second to none. Thank you for believing in me and for your supervisory role. I would like to thank Dr. Maria Liakata for providing useful comments and supervisory guide.

I would like to thank some friends and colleagues - Dr. Phillip Taylor, Dr. Bolanle Ola, Wilson Tan, Fatimah Adamu-Fika, Emmanuel Ige, Daniel Onah, Chinedu Nwaigwe, Huanzhou Zhu, Hasliza Sofian and Zhuoer Gu- for your tremendous help and friendship during my studies. I would also like to thank Pastor & Mrs. Ajutalayo and members of RCCG House of Love, Canley, Coventry - you were more than Church members, you were a family; may God enlarge you for His glory. I am grate to Mr. & Mrs. Leo Bawa, Rev. & Mrs. Kefas Tang'an - you made me a part of your families.

I also wish to thank the Texas Advanced Computing Center (TACC) for providing the Ranger logs. My appreciation also goes to PTDF-Nigeria for funding my PhD, the University of Jos-Nigeria for the support you have given me.

Lastly, but certainly not the least, I am grateful to my family- Mum, Dad, Dogara, Zumunchi, Josiah, Sirpowe, Titus and others. Your support, love, prayers and help has been immeasurable. I can not thank God enough for such a great family He has given me.

Declarations

Parts of this thesis have been previously published by the author in the following:

[53] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne. Towards detecting patterns in failure logs of large-scale distributed systems. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2015 IEEE International*. IEEE, 2015 [Chapter 4].

[54] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne. Towards increasing the error handling time window in large-scale distributed systems using console and resource usage logs. In *Proceedings of The 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA 2015)*, Aug 2015 [Chapter 6].

[55] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne. On the impact of redundancy handling in event logs on classification in cluster systems. In *Proceedings of International Conference on Dependability (DEPEND)*, Aug 2015 [Chapter 4].

In addition, the following works are under review:

IEEE-ToC An Anomaly Detection Based Methodology to Increase the Error Handling Time Window in Large-Scale Distributed Systems. *IEEE Transactions on Computers* [Chapter 6].

FGCS An Unsupervised Approach To Detecting Patterns in Failure Logs of Large-Scale Distributed Systems. *Future Generation Computer Systems (Elsevier)* [Chapter 4].

EuroSys 2016 CRUDE: Combining Resource Usage Data and Error Logs for Accurate Error Detection in Large-Scale Distributed Systems. *ACM European Conference on Computer Systems* [Chapter 5].

DSN Detection of Recovery Patterns in Cluster System Using Resource Usage Data. *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* [Chapter 5].

Sponsorship and Grants

The research presented in this thesis was made possible by the support of Petroleum Technology Development Fund (PTDF) Nigeria.

Abbreviations

APE	Average Percentage Error
BGL	Blue Gene/L
CPD	Change Point Detection
CuSUM	Cumulative Sum
DE	Differential Evolution
ECG	Events Correlation Graph
FLOPS	Floating point Operations Per Second
FN	False Negative
FP	False Positive
HAC	Hierarchical Agglomerative Clustering
HPC	High Performance Computing
HMM	Hidden Markov Model
ICA	Independent Component Analysis
JSD	Jenson - Shannon Divergence
KLD	Kullback-Leibler Divergence
LD	Levenshtein Distance
MDS	Meta-Data Server
MIC	Maximal Information Coefficient
NN	Neural Network
OSS	Object Storage Server
PCA	Principal Component Analysis
RAS	Reliability, Availability, Serviceability
RI	Random Indexing
SVM	Support Vector Machine
TACC	Texas Advanced Computing Center
TN	True Negative

Contents

Abstract ii

Dedication iii

Acknowledgements iv

Declarations v

Sponsorship and Grants vii

Abbreviations viii

List of Figures xvii

List of Tables xviii

1 Introduction 1

1.1 Motivation 1

1.2 Background 2

1.2.1 Faults, Errors and Failure 3

1.2.2 Dependability 4

1.2.3 Fault Tolerance 4

1.3 The Problem 7

1.4 The Approach 7

1.5 Thesis Contributions 8

1.6 Thesis Outline 9

2 Literature Review 11

2.1 Introduction 11

2.2 Error Detection 12

2.2.1 Unsupervised Methods 13
2.2.2 Supervised Methods 25
2.2.3 Other Methods 29
2.3 System Recovery 30
2.3.1 Checkpointing 30
2.3.2 Task Migration 34
2.4 Summary 35

3 System Description, Log Events And Fault Models 37

3.1 System Model 37
3.2 Fault Model 38
3.2.1 Categories of Fault Model 38
3.2.2 Ranger and BlueGene/L Fault Models 40
3.3 Production Systems 41
3.3.1 Ranger Supercomputer 42
3.3.2 The BlueGene/L (BGL) Supercomputer 42
3.4 System Data 43
3.4.1 Ranger Event Logs 43
3.4.2 Ranger Resource Usage Data 45
3.4.3 BlueGene/L Events logs 47
3.4.4 Definition of Terms 49
3.5 Summary 50

4 Error Detection Using Clustering 51

4.1 Introduction 51
4.1.1 Log Size and Structure 52
4.1.2 Errors and Failures 52
4.1.3 Event Logs Redundancy 53
4.1.4 Objectives of the Chapter 53
4.2 Problem Statement and Methodology Overview 54
4.3 Preprocessing 56

4.3.1 Log Events Preprocessing 56
4.3.2 Log Compression: Removing Redundant Events 58
4.4 Data Transformation 63
4.5 Sequence Clustering and Detection 66
4.5.1 Clustering 66
4.5.2 Detection of Failure Patterns 69
4.6 Experiment 72
4.6.1 Experimental Setup 73
4.6.2 Evaluation Metrics 74
4.6.3 Parameter Setting 75
4.7 Results 79
4.7.1 Runtime Analysis 90
4.8 Summary 91

5 Improving Error Detection Using Resource Usage Data and Event Logs 92

5.1 Introduction 92
5.2 Detection Methodology 94
5.2.1 Data Transformation 95
5.2.2 Event Clustering and Feature Extraction 98
5.2.3 Jobs Anomaly Extraction from Resource Usage Data . . . 102
5.2.4 Detection of Failure Patterns 105
5.2.5 Experiment and Results 105
5.3 Detection of Recovery Patterns 112
5.3.1 Introduction 112
5.3.2 Recovery Pattern Detection 113
5.3.3 Results 118
5.4 Improving Failure Pattern Detection 121
5.4.1 PCA and CPD Failure Detection Algorithm 122
5.4.2 Results 123

5.5 Summary 124

6 Early Error Detection for Increasing the Error Handling Time Window 126

6.1 Introduction 126

6.1.1 Motivation 128

6.1.2 Problem Statement 128

6.1.3 Objectives of the Chapter 129

6.2 Methodology 130

6.2.1 Root Cause Analysis 130

6.2.2 Anomaly Detection 132

6.2.3 Change Point Detection 139

6.2.4 Lead Times 141

6.3 Case Study: Ranger Supercomputer 143

6.3.1 Datasets and Performance Measurement 143

6.3.2 Base Case for Comparison - Error Detection Latency using Clustering 144

6.3.3 Identifying Anomalies Using our Methodology 146

6.3.4 Propagation Time 148

6.3.5 Other Issues 148

6.4 Summary 149

7 Summary, Conclusion and Future Work 152

7.1 Summary 152

7.1.1 Introductory chapters 152

7.1.2 Error Logs Preprocessing and Pattern Detection 153

7.1.3 Failure Sequence Detection Using Resource Usage Data and Event Logs 154

7.1.4 Increasing the Error Handling Time Window in Large-Scale Distributed Systems 155

7.2 Conclusions 155

7.3 Future Work 156

7.3.1 Improving the Error Detection 156

7.3.2 Improving the Recovery Run Detection 157

7.3.3 The Error Handling Time 157

Bibliography **158**

List of Figures

1.1 An illustration of the relationship between faults, errors and failures 3

1.2 Dependability tree diagram 5

1.3 Fault Tolerance techniques 6

1.4 An overview of the *unsupervised* detection approach 8

2.1 A taxonomy of error detection and system recovery methods . . 12

3.1 Permanent, Transient and Intermittent fault model 39

3.2 Sample Log events for Ranger Supercomputer (syslog) 44

4.1 Methodology Work flow showing the steps taken to achieve the objectives 56

4.2 Sample pre-processed logs of Figure 3.2 61

4.3 Sample preprocessed event logs (syslog) with redundant event removed 64

4.4 Event logs sequence 64

4.5 Data matrix K of N sequences, where $F_j t_i$ is the number of counts of message term t_i in sequence F_j. 65

4.6 Evaluation metrics 74

4.7 Cluster goodness based on intra-cluster and inter-cluster similarity (on Syslog, JSD metric) 77

4.8 APE (percentage miss-detection) vs Detection Threshold 77

4.9 Compression rates given varying LD on syslog 79

4.10 Showing the *F-measure* detection of both *our method* and *normal filtering* on syslog 80

4.11 Showing the *F-measure* detection of both *our method* and *normal filtering* on ratlog 80

4.12 Showing the *F-measure* detection of both *our method* and *normal filtering* on BGL 81
4.13 Showing the *F-measure* detection on both *filtered* and *redundant* logs (syslog) 82
4.14 Showing the *F-measure* detection on both *filtered* and *redundant* logs (ratlog) 82
4.15 Showing the *F-measure* detection on both *filtered* and *redundant* logs (BGL) 83
4.16 The *Precision* of our failure pattern detection and *Xu's* method on *syslog* 85
4.17 The *Recall* of our failure pattern detection and *Xu's* method on *syslog* 85
4.18 The *F-measure* of our failure pattern detection and *Xu's* method on *syslog* data 86
4.19 The *Precision* of our failure pattern detection and *Xu's* method on *ratlog* 87
4.20 The *Recall* of our failure pattern detection and *Xu's* method on *ratlog* 87
4.21 The *F-measure* of our failure pattern detection and *Xu's* method on *ratlog* 88
4.22 The *Precision* of our failure pattern detection and *Xu's* method on *BGL* 89
4.23 The *Recall* of our failure pattern detection and *Xu's* method on *BGL* 89
4.24 The *F-measure* of our failure pattern detection and *Xu's* method on *BGL* 90
4.25 The runtime graph of Detection approach 90

5.1 *Bursty faulty event sequence* showing the behaviour of fault events logged within an hour 93

5.2 *Silent faulty event sequence* showing the behaviour of fault events logged within an hour 94
5.3 Methodology work flow showing steps taken to achieve detection 96
5.4 Data matrix F_{t_w} of a sequence with N nodes and E event types, where e_i^l denotes the number of occurrences of event e_l by node n_i. 97
5.5 Jobs outlierness of a sequence using PCA 104
5.6 Evaluation metrics 107
5.7 Results showing accuracy of our detection approach under varying values of entropy threshold (φ) and $\gamma = 0.6$. 109
5.8 Results showing accuracy of our detection approach under varying values of varying anomaly threshold, with $\varphi = 0.4$. 110
5.9 Graph showing detection performance (S-measure) of *our method* and *nodeinfo* 111
5.10 Graph showing runtime performance of *our method* 112
5.11 Sequence of resource usage data 114
5.12 Data matrix M with n subsequences of S, where $x_{n,k}$ is the value of counter k in subsequence n. 115
5.13 Graph showing the change point behaviours of both recovery and failure sequences 117
5.14 Result showing accuracy of detecting recovery sequences among failure sequences using *Cumulative Sum* change point detection and varying values of detection threshold, th 120
5.15 Results showing accuracy of detecting recovery sequences among failure sequences using *KLD* change point detection, and varying values of detection threshold, th. 121
5.16 Graph showing detection performance (S-measure) of both CPD methods used 122
5.17 Graph showing detection performance of combining detection based on *PCA-anomaly* and *CPD-Recovery*. 124

6.1 Methodology work flow 131
6.2 Data matrix J with i features, where $j_{n,k}$ is the value of counter k by job n. 133
6.3 Distribution of jobs outlierness of a sequence using PCA 137
6.4 Distribution of jobs outlierness of a sequence using ICA 137
6.5 Distribution of outlierness/anomaly using MIC 139
6.6 Example showing result of CuSUM CPD on a sequence 141
6.7 Lead Time of Anomalies, Errors and Failures 142
6.8 Processing and analysis of Ranger event logs. 145
6.9 Results of clustering algorithms with two different distance metrics. 145
6.10 Distribution of outlierness/anomaly of different fault sequence for week 1 150
6.11 Distribution of propagation time for different fault sequences . . 150

List of Tables

3.1 Summary of Logs used from Production Systems 45

3.2 List of 96 Elements of Resource Usage Data 47

3.3 An example of event from Blue Gene/L RAS log 48

4.1 Summary of sequences/patterns obtained from the three production system's logs 73

4.2 Experiment Parameter Values 76

4.3 Sample Clustering Result (syslog, HAC) for a cluster with sequences (seq.1 and seq.2) 78

6.1 Sample Results of Counter correlations (MIC) 143

6.2 Anomaly Detection performance of PCA and ICA. 146

6.3 Detection performance for Change Point Detection. 147

6.4 Detection performance for Change Point Detection with PCA or ICA. 147

6.5 Distribution of Anomalous Jobs and Error Propagation Time for all the methods 148

CHAPTER 1

Introduction

Since the inception of modern computers, the performance growth has been steady and tremendous. With ever-increasing society's challenges, applications needing higher computational capabilities increase. This computational need led to the emergence of supercomputers in the 20th century. Since then, the performance of these systems have been improved to meet the growing need, leading to the era of petascale computing whereby, as at 2015, the fastest supercomputer can achieve a performance of 33×10^{15} for floating point operation per second (FLOPS)[1].

With the future exascale systems projected to have a performance of up to 1×10^{18} FLOPS, the need for these systems to remain fault tolerant becomes increasingly important [43, 76]. This is because people, key sectors or organisations are becoming more dependent on these systems. This is further buttressed by the increasing reliance on these systems by most computationally demanding applications such as used in financial systems, weather forecasting, control systems etc. The increased demand for these systems has led to continued growth in their capabilities to meet up; this has also led to their increased size and complexities.

1.1 Motivation

Attaining and maintaining a system's dependability becomes a challenge as these computing systems are not immune to failure, more so, the increasing complexity makes them difficult to manage. The failure of these systems can be costly both in terms of the time and money. This is because it takes a long

[1] http://www.top500.org/lists/2015/06/; (accessed September, 2015)

time for system administrators to identify and fix the problems. For example, the failure of applications involved in high speed financial trading can be huge.

Large-scale systems generate huge volume of “log events“ or system state data. These log events which are sometimes referred to as the health information of the system, are basically the first point of contact whenever there is a failure of the system. However, there is basically no way to know that these systems have encountered faults and could eventually fail by merely observing these system state data or event logs. This is because they are massive and doing so can be overwhelming. Additionally, only few of the events are symptomatic of faults and/or failure [103], that is, a large percentage is redundant. Fortunately, the events could form patterns that characterises failure. In other words, these event patterns preceding a particular failure are the symptoms of such failure. *Root cause analysis* methods [118], [23] have been used to find causes of failures from the event logs, however, these methods can only help find the root causes of failure, but cannot prevent it.

In order to make these systems dependable, failure runs must be detected as early as possible to enable taking other proactive failure handling methods or approaches and avoid its consequences. If symptoms of failures can be detected early, failure preventive, avoidance or mitigation measures can be applied. This thesis focuses on detection of error patterns using unsupervised learning approach in large-scale HPC systems.The logs generated by the systems at run time are often indicative of errors; these errors could eventually result in failure if not properly handled by system administrators.

1.2 Background

The concept of dependability and fault tolerance is explained in this section. We explain some basic terminologies relating to the concept of dependability in order to explain the challenge and draw home our approach.

1.2.1 Faults, Errors and Failure

A system generally is made up of interacting components. These components and/or systems are expected to display high level of dependability by delivering the expected service. However, they are usually not immune to *faults* and these faults are evident by the *errors* the system produced which could eventually lead to *failure*. These are termed as *threats* to dependability [6].

A **failure**, according to Avizienis et al. [6] "*is an event that occurs when the delivered service deviates from correct service*". That is, a failure occurs when a system transits from correct behaviour to an incorrect implementation of system's function. The anomalous system's service may assume different levels of seriousness. An **error** on the other hand, is a deviation from the correct state given that failure involves one or more system states deviating from the correct state [6]. It is important to note that not all errors would eventually lead to failure. An error is detected by the presence of a logged *error messages* by the system. A **fault** is the cause of an error. In other words, an error is a manifestation of a system's fault. When a fault causes an error, it is said to be in an *active* state otherwise, it is *dormant* or undetected. Hence, faults are the root causes of failures. Since systems are composed of components, a *failure* at the subcomponent level becomes a *fault* for the component. The concept of faults, errors, failures and their relationships is illustrated in Figure 1.1.

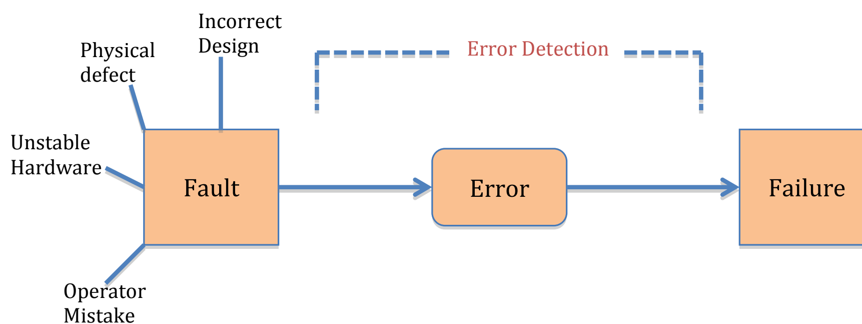


Figure 1.1: An illustration of the relationship between faults, errors and failures

1.2.2 Dependability

It is expected for systems to be dependable, that is, possessing the ability to avoid service failures that typically occur more frequently with an unacceptable severity level, according to Avizienis et al. [6]. In general, dependability is "a property of a computing system which allows reliance to be justifiably placed on the service it delivers". A system's dependability ensures that it is ready to deliver correct service (*availability*). Such a system must be *reliable*, that is, the service offered must be correct. Other attributes of a dependable systems include *integrity*, whereby a system is devoid of unacceptable alterations, *maintainability*, whereby the system can be modified or updated or repaired; and *safety* to ensure no fatal outcomes on users or the system itself.

The dependability schema showing the *attributes*, *threats* and *means* for which dependability of a system can be achieved is seen in Figure 1.2.

The **means** for which dependability is achieved is as depicted in the schema. These means include:

- *Fault prevention* which involves preventing fault from occurring, especially development faults.
- *Fault Tolerance* involves avoiding failures in the presence of faults. This is achieve by error detection and system recovery. The work of this thesis is focused on this part.
- *Fault Removal* involves diagnosis and applying preventive and corrective maintenance.
- *Fault Forecasting* estimates current and future incidences and their consequences.

1.2.3 Fault Tolerance

Fault Tolerant computing dates back to the early days of computing [140], where designs are made to withstand hardware faults. As this field emerges, both

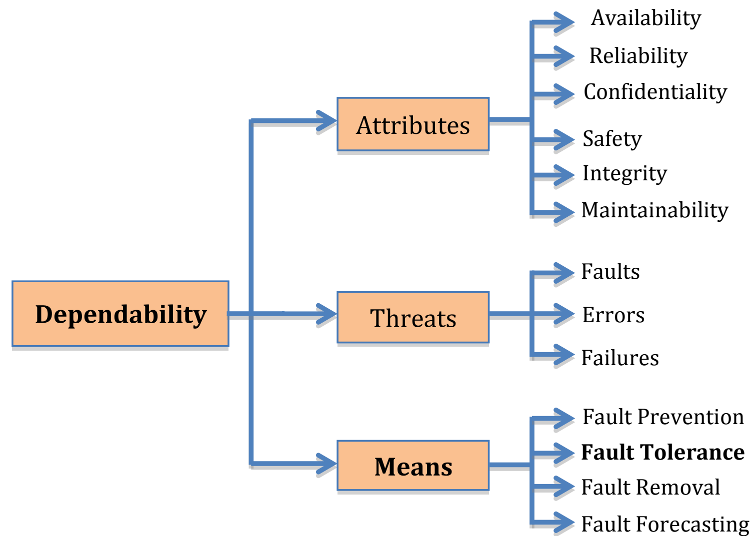


Figure 1.2: Dependability tree diagram

software and hardware fault tolerance became a focus. In distributed systems, it is expected that part of the system may be failing while the remaining part continues to function and outwardly normal. Fault tolerance in distributed systems is aimed at having systems that can automatically recover when part of the system is affected by a failure. The overall performance of the system is also not expected to be affected. In other words, the system is expected to continue its normal operation at an acceptable level even in the presence of failure.

Fault tolerance approaches are aimed at failure avoidance [4, 5]. The processes involved in achieving this are *error detection* and *system recovery*. For clarity, we show the techniques involves in a schema diagram shown in Figure 1.3.

Error handling techniques are aimed at eliminating errors from the system's state. Such methods include rollback recovery techniques (e.g. checkpointing) and rollforward techniques (where system's state without errors is used as new state). Fault handling techniques are targeted at preventing future occurrence of failure. Root cause analysis techniques belong to this category.

Error detection methods identify the presence of an error in a system. These methods are implemented to detect errors either during the system's normal

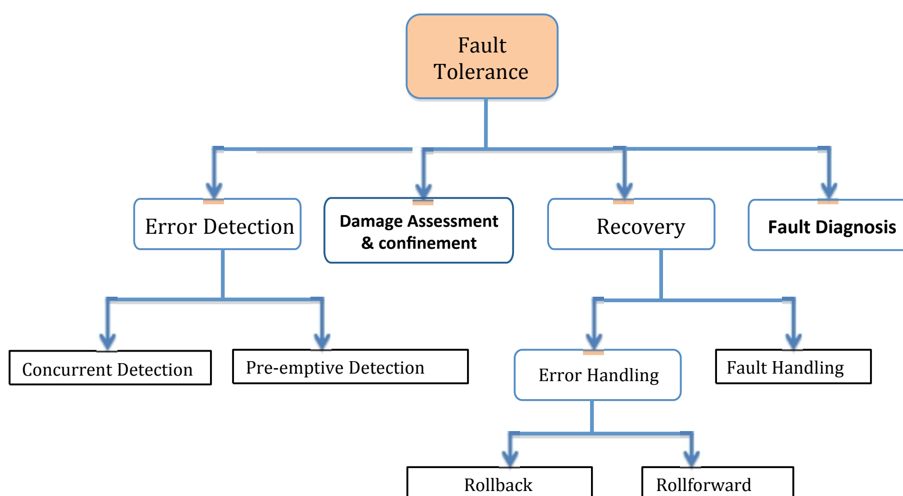


Figure 1.3: Fault Tolerance techniques

operation or when it is suspended. Usually, error detection is performed first before rollback recovery or rollforward techniques are invoked for any recovery process.

As explained earlier, *faults* are manifested through errors and these errors are logged by systems as *error logs*. The errors are pointers to eventual failures, hence, these error events are patterns or signatures of such failures. In this work, error detection is sometimes called *failure pattern detection*. When an error always lead to a failure, then *error detection* in this case, is similar to *failure prediction*. This is because when an error is detected, the likely future occurrence of failure can be predicted easily, in particular, error detection is first performed to enable prediction. Hence, we use the two terms to mean the same. Similarly, in this thesis, the term *fault detection* implies detecting the manifestations of faults, which are errors. Hence, we sometimes use *fault detection* to mean *error detection*.

1.3 The Problem

Large-scale HPC systems produce a significant amount of data (error events and resource utilization data), and these logs contain information about the system's activities. Whenever a fault occur, errors may become visible and these errors can take a particular pattern depending on the fault [43]. The systems can undergo updates which can change the nature and behaviour of faults and the patterns of error events produced. Similarly, abnormal activities in the system can be experienced due to these faults. The abnormality can be seen in the way the system resources are being used, which is captured in the resource utilization data. If the system administrator does not act upon the errors, an error or some can lead to system failure.

The research questions this thesis seek to answer include: first, can the huge event logs be reduced by filtering redundant ones and still contain useful ones for failure analysis? Secondly, how effective is using event logs and resource utilization/usage data for error detection? And can errors be detected early even when it is not visible yet to enable early prevention of failure?

1.4 The Approach

In this thesis, we develop an approach to error detection based on the analysis of event logs. Owing to the properties of the systems and the nature of the data available (huge, unlabelled and semi-structured), we chose an *unsupervised approach* to error detection. In this method, the characteristics of the system based on the normal and the rare abnormal behaviours, error detection is made without having a prior knowledge of the patterns. A *supervised* learning approach [115] requires an adequate knowledge of the system, its faults and failure patterns and properly labelled for training to take place. This can be difficult given that system updates usually change these patterns and manual labelling of the data can be overwhelming even for experts. Unlike supervised learning method, the *unsupervised* approach can still capture these patterns even in the

presence of updates and without the need for complete expert knowledge and labelling of the event data. The fact that failures are rare events makes an *unsupervised* approach relatively more viable as the problem can be seen as an *anomaly detection*. The general overview of the unsupervised approach is depicted in Figure 1.4. It involves extracting features from sequences of event logs and usage data in order to determine if a sequence is failure inducing or not. In some cases, a preprocessing of the data is required.

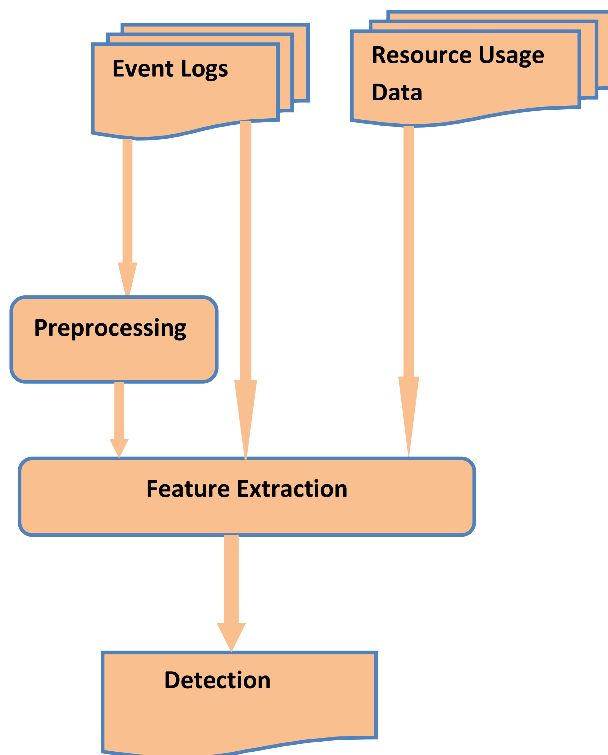


Figure 1.4: An overview of the *unsupervised* detection approach

1.5 Thesis Contributions

This thesis is focused on achieving fault tolerant systems with particular contributions to development of a novel approach for error detection and recovery in computer systems. We performed experiments on three different data sets

from two production systems and the results demonstrated excellent performance; outperforming established error detection methods in the field of fault tolerance. We make the following specific contributions in this thesis:

- We proposed a *novel and generic* approach to log filtering that not only filters redundant events, but also preserves events that are not similar but causality-related. This preserves event patterns that can serve as precursors to failures.
- We proposed a novel clustering-based failure pattern detection approach that utilizes the inherent characteristics of faults to detect the presence of errors in computer systems.
- A novel method for error detection based on the combination of *event logs* and *resource utilization data* is proposed. This method uses the anomaly in the use of computer resources and the informativeness of event patterns to detect errors in the system.
- We propose an approach for detecting *recovery patterns* in system data. These are patterns which characterise errors which do not eventually lead to failure. The approach, which is based on change point detection, identifies such patterns based on the consistencies in which resources are utilised.
- A novel method for increasing the *error handling time window* is proposed. This is the time window for which error handling techniques can be applied and it must be large enough for such techniques to complete. Our method increases this time by detecting errors early from minimal error symptoms.
- In this thesis, we provide a taxonomy and a comprehensive survey of fault tolerance techniques.

1.6 Thesis Outline

The remainder of the thesis is structured as follows:

Chapter 2 contains a taxonomy and overview of the current work in the field of dependability in large-scale systems. Particularly, the chapter describes fault tolerant methods with error detection approaches categorised under supervised, unsupervised and other methods. The chapter also reviews system recovery methods.

Chapter 3 briefly describes the system model for which the approaches can be applied. This chapter gives an overview on two supercomputing systems (Ranger and Blue Gene/L) and the data collected from these systems for the experiments performed. It further explains the fault model assumed in this thesis and some basic terminologies related to the use of the data and the approaches used.

Chapter 4 presents a filtering method used to reduce the redundant events in the log data. The chapter further details a clustering approach for detecting error event patterns from these logs that are symptomatic to failure.

Chapter 5 focuses on utilizing both resource utilization data and event logs to detect failure-inducing behaviour in systems. The chapter details the approach which obtains anomalous behaviour in systems from resource utilization data combined with the nodes' behaviour captured by the event log entropies to detect the error patterns. The chapter explores other methods of improving failure pattern detection for which recovery pattern detection was proposed based on change point detection.

Chapter 6 details the approach for improving the time for which error handling can be performed. This stems from the fact that with future systems projected to have reduced mean times to failure, it is necessary that identification of failure leading errors should be done early enough to enable error handling techniques to complete successfully.

Chapter 7 summarises and concludes the thesis. The chapter discusses the implication of each approach presented as well as their limitations. The chapter also provides directions for future work and further research.

CHAPTER 2

Literature Review

2.1 Introduction

The complexities associated with large-scale systems are often challenging; failures become an everyday challenge to deal with. Determining the causes and impact of these failures can be elusive sometimes. With increasing and proven approaches to preventing faults from resulting in failures, it becomes important that symptoms of failures (errors) manifesting as a result of the presence of faults in a system are detected on time. In fault tolerance, successful error detection provides a platform for system recovery to be invoked in order to prevent failure. In this chapter, we survey different published work that relates *error detection* and methods used for performing *system recovery*. This survey follows the taxonomy illustrated in Figure 2.1. We discuss methods for error detection in which they can be categorised as supervised, unsupervised or other methods. By other methods, we mean those which can not be clearly categorised under supervised or unsupervised, or are the combination of both. We discuss error detection methods in Section 2.2. Section 2.3 contains system recovery methods. We summarise and conclude this chapter in Section 2.4.

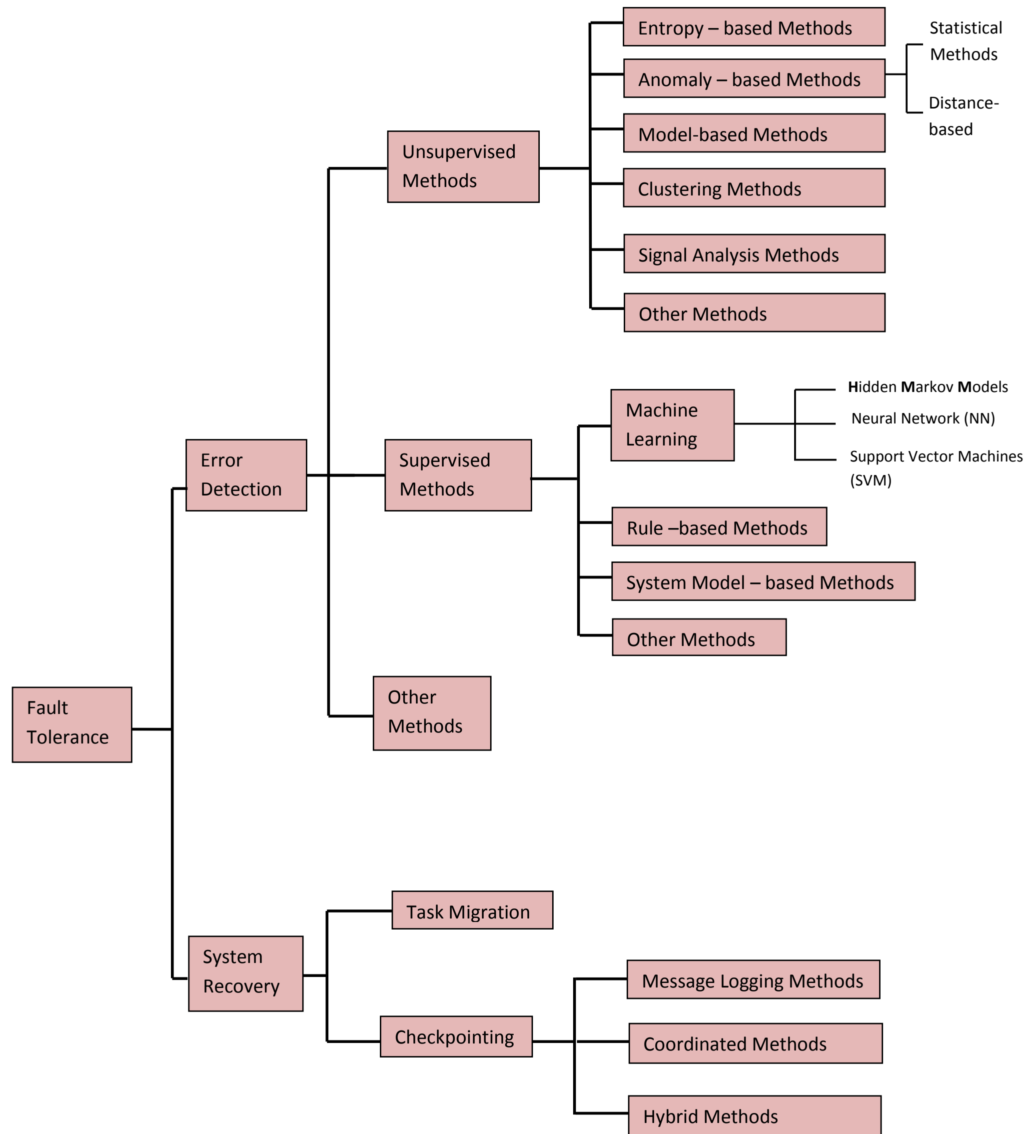


Figure 2.1: A taxonomy of error detection and system recovery methods

2.2 Error Detection

We discuss major approaches and related work on error detection. These methods are categorised into two: unsupervised or supervised approaches. By supervised, we mean methods for which learning (training) of the error patterns

is done from known data before testing is performed on unlabelled data. Unsupervised method on the other hand, are methods for which detection is done without any knowledge of previous patterns.

2.2.1 Unsupervised Methods

Detecting error patterns for any proactive failure avoidance measure has received much attention from researchers. Learning such patterns in an unsupervised way has proven to be effective. Generally, these techniques share a common purpose: detecting faults and/or failure inducing errors in systems. These techniques may differ in approaches; we explain the frequently used ones.

Entropy Approach

An entropy-based approach to detecting failure inducing patterns has also been shown to be successful in the domain. It involves capturing the entropy of the system from the event logs. Particularly, several approaches have been developed [106] [94] to detect alerts in logs of large-scale distributed systems.The concept of entropy has also been demonstrated to be useful in detecting changes in the behaviour of distributed systems components [26].

Basically, entropy-based detection methods [106], [93] based its motivation on the premise that in large-scale distributed systems, similar computers correctly executing similar jobs should produce similar logs. That is, they should produce similar content or line tokens. The method leverages the information that each line token carries with respect to the node that produced it. The approach is as follows: Given W as a set of unique terms (tokens) formed by concatenating line tokens with its position in line, C is the total number of nodes, then the matrix $M = |W| \times C$, with $x_{w,c}$ as the total number of counts of term w appears in messages generated by node c. To capture the uneven

distribution of the terms among nodes, given G vectors of $|M|$ weights, then,

$$gw = 1 + \frac{1}{{log_2}^C} \sum_{c=1}^{C} p_{w,c}\ log_2(p_{w,c}) \tag{2.1}$$

where $p_{w,c}$ is the number of times term w occurs on nodes c divided by the number of times it occurs on any other node,

$$p_{w,c} = \frac{x_{w,c}}{\sum\limits_{c=1}^{C} x_{w,c}} \tag{2.2}$$

The next step is to obtain the informativeness of the nodes, *Nodeinfo*, within *nodehours* (all event lines within an hour by a node) and rank them based on how many high information terms are contained in each. Hence, let H be the set of all *nodehours*, and let Y be the $|W| \times |H|$ matrix where $y_{w,c,j}$ is the number of times term w occurs in nodehour H_j^c, then the Nodeinfo for each nodehour is given as:

$$Nodeinfo(H_j^c) = \sqrt{\sum_{w=1}^{|W|} (g_w log_2(y_{w,c,j}))^2} \tag{2.3}$$

where each nodehour is ranked based on decreasing value of Nodeinfo. Those with high Nodeinfo are considered to have alerts(errors) [106]. This approach was applied on logs of supercomputers with good result.

Another entropy based approach is proposed in [94]. This approach characterises system behaviour from system logs. Similar to Oliner et al. [106], this approach leverages the entropy-based information content of the logs within a spatio-temporal partition. A clustering is applied to the information content of these partitions to characterise failure and normal behaviour. In particular, the information content values corresponds to the entropies of the message types within a spatio-temporal partition. These entropy values serve as attributes to the clustering algorithm. The clusters formed are assumed to describe different internal states of the system for which a failure inducing or normal state is detected. The authors argued that the message types provided a better perfor-

mance rather than using every field of an event log. In [53] the authors utilise entropy of events logs and event characteristics to detect failure patterns in logs.

One of the advantages of entropy - based methods is that they can be implemented easily so long as features can be captured accurately. However, the approach suffers from the following disadvantages:

- Since it depends on obtaining the informativeness of terms of the log events, it can be computationally expensive as the increase in the number of the terms, the higher the size of the features to be considered.
- Its performance is dependent on the nature of logs, that is, it performs poorly on logs where faults are not characterised by presence of high event messages.
- It requires system logs to be decomposed into spatio-temporal partitions; the act of partitioning may not capture the desired events with the required time window or partition, hence decreasing the detection accuracy.

Anomaly-based Approach

The challenge of error detection can also be seen as anomaly detection problem. Anomalous pattern detection assumes the existence of a normal pattern for which an abnormal one can be viewed as a deviation from the former. We discuss some of the approaches widely and recently used under different categories as follows:

1. **Statistical anomaly detection methods:** Statistical approaches take two steps, the training and the test stages. At the training stage, a statistical model is built based on the historical data; this is used for both supervised and unsupervised anomaly detection; here we explain the unsupervised approaches. Similar to work in [67], Das et al. [27] proposed an approach that scans through categorical data and detects a subset which is thought to be anomalous. In this method, groups of records rather than

individual records are assumed to substantially increase detection. It calculates the anomaly score of the subset of the records. The anomalousness is the maximization of anomaly scores over all subsets where those that deviates abnormally from others are likely anomalous. This is achieved by learning a Bayesian network model over the training data where the conditional probabilities of each attribute of a record are calculated forming the anomaly scores. This approach was applied on categorical data. Similar approaches can be found in [151]. Most of these techniques assume the knowledge of the underlying data distribution. Even though this method was not used for detection of errors, it can be adopted easily. The logs which are the records can be categorised or other to obtain their anomaly scores.

The frequency or histogram approach is a non-parametric method that constructs a histogram of normal and anomaly data. Particularly, the feature-wise histogram is constructed where by, at the test stage, the probability of each feature is obtained. The anomaly score is then calculated by summing the probability values of each feature, which is calculated as follows:

$$anomaly_score = \sum_{k \in X} w_k \frac{(1 - p_k)}{|K|} \tag{2.4}$$

where w_i is the weight assigned to the feature k, p_k is the probability distribution of feature k and K is the set of features.

This approach is well-applied in fraud detection [35], intrusion detection [148], [149].

Another work that statistically models the normal behaviour of systems in order to detect failures as anomaly is reported in [19]. That is, the normal system's behaviour is modelled for the internal measurements of the systems to be tested against any deviation. This method utilizes subspace mapping between system inputs and the internal measurements to model their dependencies. It further leverages on the statistical method, canon-

ical correlation analysis (CCA) to discover highly correlated subspaces between the sets of variables. The authors further proposed a method called *principal canonical correlation analysis* to capture the variance between the two features (system inputs and internal measurements). The authors reported good detection based on the deviation observed of internal measurements from the system normal models.

In [149], the authors describe an approach that detects outliers in categorical data. The method addresses the problem using a statistical learning approach which is an online unsupervised learning process of a probabilistic model. The method performs online learning and updating as follows: Whenever a data point input is given to the system, it learns the probabilistic model using an online discounting learning algorithm. The input data point is then scored based on the learnt model; with high score indicating a high chance of it being an outlier. This approach was implemented and tested on network intrusion detection data for KDD cup 1999 and the rare event detection for pathology dataset of the Australian Health Insurance Commission. The authors recorded good detection.

2. **Distance - based:** These methods detects failure patterns based on the dissimilarity and rarity of certain inputs from others. A dissimilarity threshold is defined in this case. We discuss some distance-based anomaly methods as follows.

   Xu et al. [147] formalised problem detection in systems as an anomaly detection problem. Their assumption for this is that an unusual occurrence of error log message is often an indication of a problem. However, a single event would not be sufficient to point to a problem, and even groups of events may not, but relationships among events can. In the approach, features are created to capture the relationship among events for possible identification of abnormal ones using the source codes and logs. It creates features that succinctly capture the correlation among the log messages

and perform detection using PCA extracted anomaly vectors from the features. The approach used on console logs is summarised as follows: Parse logs and source codes to extract useful schema, then create features that capture correlation among events (state ratio vectors and message count features). This is done by combining source codes and message logs where the hidden log schema is extracted. This feature vectors are used as inputs to the PCA and detection algorithm. In employing PCA for anomaly detection, the feature vectors are labelled as normal or anomalous. The intuition is that, highly correlated feature vectors can be identified and anomalies are assumed to be rare. Therefore, feature vectors that deviate from the correlated pattern are likely to be anomalous. To identify abnormal vectors, a distance from the uncorrelated subspace to the normal subspace is calculated to determine the abnormality of a vector. That is, a vector v is projected onto an abnormal space v_a, the squared distance is the squared prediction error (SPE) calculated thus:

$$SPE = ||v_a||^2 \tag{2.5}$$

and $v_a = (1 - AA^T)v$, where $A = [a_1, a_2, ..., a_k]$ is the k formed principal components by PCA.

Since it was assumed that the abnormal vectors are distant from the normal subspace, then detection of abnormal vectors is simple: flag v as abnormal if SPE is greater than certain defined threshold.

This method was shown to have good detection rate. However, it requires access to source codes of programs, which are not readily available. PCA computation can be costly; hence this might not be suitable for an online detection.

Lan et al. [80] present an anomaly based detection technique that finds the abnormal nodes in large-scale distributed systems. The approach consists of three steps for detecting the abnormal nodes using system data.

The first step involves transforming the data from multiple data types to a single one to enable detection. Specifically, it involves converting variable-spaced time series to constant-spaced ones, removing noise etc. The result of this step is a feature matrix formed for each of the nodes. The second step deals with feature extraction; where principal component analysis (PCA) and independent component analysis (ICA) are used to extract useful features to be used for detection. Lastly, the abnormally behaving nodes (outliers) are identified using a proposed outlier/anomaly detection algorithm. In order to identify the nodes that are anomalous or significantly dissimilar from others, the authors utilised Euclidean distance. Hence, given two data points $y_a \in \Re_s$ and $y_b \in \Re_s$, then, distance is given by:

$$d(y_a, y_b) = \sqrt{\sum_{i=1}^{s} (y_a, i - y_b, i)^2} \tag{2.6}$$

Therefore, the nodes further away are anomalous. The authors demonstrated that the method is able to detect anomalous nodes with high specificity.

In our method, entropy is extracted as one of the features of event sequence. We combined with resource usage data for which an anomaly score was obtained. Detection is achieved based on features extracted and the anomalousness of the resource usage of the system.

Clustering Methods

In this context, clustering is aimed at grouping similar inputs patterns together. It is believed that those that are likely to be failures are rare, hence detection can be performed, sometimes, in conjunction with other methods.

Fu et al. [37] developed a method that exploits spatial and temporal correlations for failure prediction in coalition of systems. In this work, the authors first identified failure signatures form the running system and then developed a covariance model that can adjusted in time to measure the temporal correlation

of events and further show their spatial correlation using stochastic model. Both temporal and spatial correlations are used to model failure propagation in the system. They further developed a clustering algorithm that groups signatures based on the correlations observed. The clusters formed are used to predict future occurrence of failure.

In [119], a clustering-based approach is proposed for detecting abnormal events. They achieved this by first, extracting relevant features and then proposed a proximity clustering that groups patterns of events based on the their semantic relatedness. The sparsely clustered features are believed to be anomalous or abnormal. This approach was used to detect abnormal events from a surveillance cameras system. It can be adopted and used for detecting failures in logs of large-scale distributed systems. This can be done by creating useful features from the logs that describes correlation among events; then applying clustering to separate normal from abnormal features. Another work that employs clustering for detecting failures in large-scale distributed systems is found in [82]. The work used clustering with gossip-base algorithm to perform this. The clustering groups messages based on location while the gossip-based algorithm is aimed at removing uncertainty from massive logged messages and also reduce the detection time. The authors did not elaborate on the performance of the detection approach but demonstrated how it can incorporate existing methods to reduce detection time. Our clustering approach is different; we obtain clusters of similar events to enable extraction of features of patterns particularly entropy and mutual information. These features are then used for error detection.

Signal Analysis Methods

Gainaru et al., [43, 44] proposed a hybrid method that uses signal processing and data mining techniques to predict faults using logs. This approach is motivated by the fact that the behaviour of event types of HPC systems and how the behaviour is affected by errors overtime can be captured. The hybrid approach

characterises event types and detect faults and normal patterns in event logs. The approach extracts and represents the event types as signals where these signals are appropriately characterised and an anomaly detection method is applied to identify anomalous signals. The authors went further to implement this this as online outlier detection. The offline event correlation/analysis is combined with online monitoring and detection of outliers for identification of any deviation from known normal signal patterns for prediction of possible failures. The authors reported good results for predicting failures.

In [102, 105, 107] the authors proposed methods that can address several problems in production systems. Such problems include: abnormal interactions among components and identifying these bad behaviours of systems. The components behaviour is captured as “surprise”; that is, measuring how anomalous a component is. These anomalies (surprises) are captured as anomaly signals. The anomaly signals are obtained by finding deviations from known models of normal component behaviour. In this method, the degree of anomalousness of each signal is retained rather than discretizing them into either abnormal or normal. In particular, the method represents behaviour of components as signals and computes the anomaly score for each to be able to identify abnormal behaving components. The anomaly score is computed by comparing the histogram of a recent window of component behaviour with the entire history of behaviours of the components. Kullback Leibler divergence [77] is computed between the probability distributions of the observations. This provides the distance or how each distribution differs from the other. Those with high deviation from the normal are considered anomalous signals.

The authors went further to construct a Structured-Influence-Graph (SIG) from the anomaly signals. This graph shows the correlations between components. Even though these methods did not target the detection or prediction of failures, the steps are easily adopted for this purposes as explained by the authors. The models of component behaviours can be used to predict any future occurrence of failure. The conversion of these behaviours to signals and

the using anomaly detection to obtain anomalous signals is indeed a good way to detect faulty components in large-scale systems. The online version of this method is detailed by the authors in [102].

Rule-based Methods

[38] explored a rule-based method of predicting failure events in logs of large-scale systems. They mine correlations in events by leveraging on the unique characteristics of the events. The approach first proposed a new algorithm called Apriori-LIS that mines rules from events representing correlation among them. These rules are then represented using a proposed Events Correlation Graphs (ECGs). The prediction algorithm is built based on the ECGs where the probabilities of failure events are calculated based on correlation seen in the vertices of the ECGs. The authors reported good prediction results when applied on logs of production systems. One major advantage of this approach is its ability to generate rules based on events correlation for effective detection.

Model-based Methods

Cormode [26] introduced an approach that models continuous distributed monitoring of streams of data in a distributed computer system. A function for these streams of observation is computed. This method can be used for monitoring usage of compute nodes for detection of abnormal usage patterns of the nodes. These abnormal usage patterns are pointers to failure.

In their work [40], Gabel et al., proposed an approach for detecting faults from large-scale systems. The authors hypothesized that system failures are not caused by abrupt changes, rather, by the resultant effects of long time systems performance degradation. This idea is contrary to the norm, where abrupt changes are seen as pointers to failure, even though this hypothesis may have some support from work in [120]. The authors argued that machine behaviour that is indicative of faults presence would eventually result in failure. Hence, they developed a framework that uses standard numerical counter readings of

the different machines to compare those performing similar tasks within the same time frame. Anyone that significantly deviates from the others is tagged suspicious. This idea is similar to those presented in [74, 106]. A statistical model is used to capture the behaviours of the machines. The authors implemented the detection framework using three test algorithms: the Sign test [30], Tukey test algorithm [133] and the Local Outlier Factor algorithm [12]. The authors reported a good detection of outlier machines.

In [62], models of different hardware component failures is constructed from a 5 years logs collected from a HPC system. The failure models are based on each components usage and capture the correlation between components. The authors demonstrated that these application-centric models are useful in performing other system reliability methods like checkpointing. These models can be utilised easily for performing detection of these failing components.

Log Filtering and Error Detection Methods

Filtering or pre-processing logs for failure analysis is an important process that is done for any proper log analysis [88]. It eliminates redundant events from logs while keeping the useful ones or those patterns that are important for failure analysis. The normal log filtering [88] approach removes repeated events within certain time window. Specifically, similar events that are logged in sequence within a defined time window are filtered with only one kept. The challenge with this method is that, it can remove fault events that are relevant for analysing causal correlations among events. Zheng et al. [154] proposed an approach that can filter causality-related events or what they termed ‘semantically redundant‘ events. Their idea is that events may be different but may always co-occur together. This co-occurrence is not normal, hence they believe such events have similar root-cause and can be filtered. In their approach, redundancy from both temporal and spatial viewpoint is considered. This method can preserve useful patterns and was shown to be better than normal filtering. However, the methodology is log-specific: that is, it is dependent on the severity level of logs.

Most production systems' logs do not have that. A general approach is more suitable.

Unlike theirs, this work assumes that temporal events must occur in sequence to be removable and we believe that causality-related but semantically unrelated events are patterns or signatures to failure, therefore, are not filtered.

Gainaru et al. [42] proposed a log filtering approach to enhance failure detection. This method clusters similar events that tend to occur massively in sequence. In addition to clustering, Makanju et al. [95] went further to index (IDs) the result of the clustered logs for easier use by any log analysis algorithm. these clusters represents the different event types of the logs. The authors mainly focus on extracting the message types that can be used for indexing, visualization or model building. One of the caveats of this approach is that it clusters events/message types that are believed to have been produced by the same *print* statement and their occurrences is non-overlapping. Another approach that use clustering can be found in [70]. By contrast, the method in [55] can cluster non-overlapping events together. In [3], two algorithms for discovering patterns in systems event logs we proposed. The first is the text clustering algorithm which automatically discovers templates generating the messages. The second algorithm discovers patterns of messages in the system. The clustering algorithm focuses on creating a dictionary of event types from the text messages with no aim of discovering any faulty pattern, but only different patterns that can be discovered. In [53] the authors went further by discovering and detecting patterns that are symptomatic to system failure. The authors group similar patterns through clustering based on the events similarities.

In their work, Gainaru et al. [41] [43] proposed a method for analysing systems' generated messages by extracting sequences of events that frequently occur together. This provides an understanding of the pattern of failures and non-failure events. Their approach uses dynamic time window. They also use signal analysis technique to characterise the normal behaviour of system and how faults affects it; in this case, they will be able to detect deviations from the

normal.

[114] developed a method that ranks log messages deemed useful to the users. Their approach assumed that most frequently occurring messages are ranked higher using an unsupervised algorithm.

Another approach that ranks logs for failure detection is [129]. Similar to [106], the ranking is based on the location of a text word in a message and similarity in the workload performed by computer nodes where they are assumed to have generated similar log messages. They demonstrated that the method with information entropy of logs produces a very low false positive rate (0.05%) for failure event detection. One of the challenges of this method is that it may not detect failures characterised by presence of few events since it will receive low ranking.

In their approaches,[23] [24] produced a diagnostic tool *FDiag*, which uses the combination of message template extraction, statistical event correlation and episode construction to identify significant events that led to compute node soft lockup failure in Ranger supercomputer system. Events which are highly correlated with the failure are extracted as episodes.

2.2.2 Supervised Methods

Supervised detection approaches learn the patterns of errors or failure inducing and nonfailure inducing through a supervised training algorithm. After learning from the labelled data, it produces models that can be used to classify, from test data, the patterns identified as failure and non-failure by the model.

Machine Learning

Machine learning methods performs function approximation where the target values are represented by the function mimicking the input data. The target function could be the probability of a failure occurrence. Hence, the output of the target function is a boolean indicating either it is a failure or not. Another category is classification where the target is mostly binary values. It involves

coming to a decision without having to build a target function. We survey papers that use machine learning in the section.

The authors of [99] combined simulation models and machine learning method to perform detection of faults. Unlike other normal machine learning approaches reported in [100], where failure and non-failure patterns are detected, this approach performs classification of more categories of failure. It first, simulate models of the system, then using Neural Network method, it learns different faulty and non- faulty condition from the model signals. This method was validated on inverter-monitor systems and not distributed computer systems. However, this approach, similar to [100], can be adopted for failure detection in the cluster systems. More machine learning methods that were used to detect hard drive failure are detailed in [132]. They modelled the problem as a sequence labelling challenge, where, a Hidden Markov Model (HMM) is used to model these failures as probabilistic models. They extracted 68 features from hard drive data where HMM is trained for each of the supported feature. The time series Self-Monitoring and Reporting Technology (SMART) data is labelled as either *fail* or *good*. Yamanishi and Maruyama [148] came up with a different approach to HMM. In their approach, dynamic systems behaviour is represented using a mixture of Hidden Markov Models; these models are learned using an online discounting learning algorithm. Failure pattern detection is done subsequently based on anomaly score given to observed patterns. This approach basically utilises the HMM for learning the models.

Other HMM-based methods for error detection similar to [148] is detailed in [153]. In their work, Murray et al. [100] compared several machine learning techniques for detecting or predicting failures in hard drives. They formulated the problem as a rare event detection in time series data of noisy and non-parametrically distributed data. They developed a detection algorithm based on many other learning frameworks and Naïve Bayesian classifier to perform detection. Support vector machine (SVM) was also used. Experiments demonstrated that SVM, a hyperplane separation technique outperforms the Naïve

Bayesian classifier implementation.

Fulp et al. [39] proposed a new spectrum-kernel SVM approach to predict failure events from log files of computer systems. The approach extracts messages in a sub-sequence or sliding window to predict likely failure. The frequency observation of the messages in each sub-sequence is used as an input to the SVM. The SVM then classifies the messages as either failure or non-failure. Other methods that combine SVM and other classifiers for detection can be found in [118, 152]. The work in [152] first extracts sets of features that accurately captures the characteristics of failures. The authors then investigated four classifiers (rule-based, SVM, Nearest Neighbour and a customised Nearest Neighbour) to detect failures. They reported good results when applied on an IBM BlueGene/L log data, achieving better performance with the customised Nearest Neighbour method.

Fronza et al. [36] presented an approach that predict failure of systems using log files based on random indexing (RI) and Support Vector Machine (SVM). They presented a two-step approach where, firstly, given labelled (failure and non-failure) sequences of logs files representing a change in system state, a feature vector is constructed. These features are created using RI. It is a word space model that accumulates vectors based on occurrence of words in context. This approach does not require the data to undergo further dimensional reduction since it forms this vectors incrementally. Secondly, SVM is applied on these data to separate or classify failures and non-failure patterns based on models formed. This approach was validated on an industrial data and the authors reported low performance on true positive rate and high true negative rate.

Neural Network (NN) method has long been used in error detection [51]. In his work, Niville [101] explained a standard neural networks method for detection of failure patterns in large-scale engineering plants. A fully connected cascaded neural network approach was described in [64] for detecting or identification of sensor failures of aircraft. The authors reported that out of 150 experiments conducted, only one fault went undetected. A similar work has

been done and presented in [51], except that, this method uses the traditional NN. Similar work have been proposed for detecting failure in automotive engines [144]. However, the authors were able to verify their technique by simulation. Similarly, Selmic and Lewis in [121] presented a neural network approach that identifies and detects fault in nonlinear systems. The approach used however, is a multimodel NN. An NN of the system that emulates the behaviour of system is trained offline based on known nonlinear models. At the simulation part, the neural net is compared to the real non-linear plant for detecting possible failure. The authors did not show detection accuracy but demonstrated how the estimation error converges asymptotically. An approach that combines models of support vector machines and back-propagation neural network is detailed in [155]. This method was aimed at improving drive failure detection in large scale storage systems. The models are tested on a real-world dataset and was shown to have high detection accuracy. These methods can be used for detecting failures in computer systems if models of the system failure signatures can be obtained and trained using NN.

Lee et al. [83] presented an approach that captures the contents of fault trees and detect faults using decision trees. A decision tree is built and trained from the sample data containing faulty and non-faulty events. The authors presented the result of such classification as a diagnostic decision tree. This tree helps reveal unknown fault paths also.

Rule-based

In the context of supervised approach, rules are generated which reflects normal and faulty states of the computer systems. These rules are used to detect failure patterns from system inputs. [139] described a rule-based method that first characterised the target events by observing those that frequently precedes it within a given time window. An algorithm is proposed that searches these targets and preceding events in order to generate rules describing the behaviour of the patterns. The authors further introduced *association rule mining* [2]

and classification with consideration to the time of occurrence of events. This combination enabled the building of robust rule-based models that can be used to detect or identify rare events. The authors reported that the time window affects the performance of this method.

System Model-based

Model-based detection has long received attention [143] [47]. The authors of these papers surveyed works that utilised models to detect and isolate failures in dynamical systems and production plants respectively. A mathematical model of the system and its problems (e.g., memory leaks, sensor biases, equipment deterioration) are detected when there is deviation from the normal system model signatures. [7] developed a Markov Bayesian Network model for predicting failures. This method combines causal information and updates the model estimator with new observations.

Vaidyanathan and Trivedi [137] proposed a semi Markov reward model (measurement-based) to estimate the rate of exhaustion of operating system resources. The authors utilised statistical cluster analysis to identify the various network workload states. This is done through clustering of the measurements taken. Next, the authors build a state-space model and then define a reward function based on the rate of resource exhaustion corresponding to each resource. The state reward defines the rate of change of the modelled resources. In order to estimate the rates, a linear function is fitted to the data. This approach estimates the time until resource exhaustion by computing the expected reward rate at steady state from the semi Markov model.

2.2.3 Other Methods

Methods that combine both supervised and unsupervised are highlighted in this section.

Yamanishi et al. [148], concerned with the challenge of detecting outliers from unlabelled data, propose a method that combines both supervised and

unsupervised approaches to detect outliers. The authors perform detection as follows: First, a method [149], an unsupervised online detection of outliers is applied to the data. This method obtain an anomaly score for each data point. A high anomaly score indicates that the data point is likely an outlier and lower scores indicating less possibility of an outlier. The data is then labelled based on the scores with higher scores labelled as "positive" and low scores labelled as "negative". From these labelled data, a supervised learning method is used to generate outlier detection rules. These rules are generated using stochastic decision list (used as classifier). For the selection of the rules, it utilises the principle of minimising extended complexity. Failure detection is then done using the generated rules on any input data. With the use of system logs, a proper method for extracting patterns that can capture the outlierness of the input sequences can be used, then an anomaly score is assign to each sequence. Finally, a classification algorithm can be used to generate rules to be used for detection.

2.3 System Recovery

The processes involved in achieving fault tolerance include error detection and system recovery. Rollback, rollforward techniques like checkpointing, job migration etc., are applied whenever a fault is detected.

2.3.1 Checkpointing

A checkpoint is an identified area in a program at which fault free system states or status information that would allow restarting of processes at a future time is saved. The process of saving system state and status information is called checkpointing [73]. Periodically, checkpointing process is done and this checkpoint become points of restart whenever a failure noticed [73], [32]. We explain some few checkpointing methods used in distributed systems.

Checkpointing in current large-scale clusters performed on disk under the

control of I/O node is limited by bandwidth of that I/O node. According to [15, 45], coordinated checkpointing would not scale for future exascale systems because, their design is centred on energy efficiency. Furthermore, El-Sayed and Schroeder [33] provide an extensive study on the energy/performance trade-offs associated with checkpointing and pointed out that periodic checkpointing methods would not scale perfectly in exascale systems [45, 110] and requires further improvement. The best option possible in this case is to save checkpoints on local nodes. However, the disadvantage of this method is that it can only recover from software faults, since nodes with checkpoints could also fail. Preventive checkpointing methods are promising as they avoid the earlier mentioned challenges. In this methods, prediction is done based on successful detection of non-fatal precursor events. In a sense, early effects of faults must be detected in order for checkpointing to be triggered. Some approaches [21, 29, 57] circumvents checkpointing to provide fault tolerance.

Checkpointing in Distributed Systems: Checkpointing is useful, however, it is also more difficult in distributed systems due to the fact that there is no global clock to bring synchronization to checkpoint streams. The checkpointing methods are discussed under the following:

1. **Message Logging Techniques**

   In this technique, processes save their states independently and logging of inter-process messages [141]. Most of the checkpointing algorithms based on message passing techniques are a variant of Candy & Lamport algorithm [18]. They proposed an algorithm that takes a global snapshot of distributed systems. It assumes a distributed system contain a finite number of processors and channels. The algorithm builds the global state of the system by harmonizing the processors; the states are logged at checkpointing time. A marker messages are used to identify those coming from different checkpoint intervals. According to the authors, a central node first initiates the checkpoint algorithm and then other nodes follow after receiving the special marker message. Other variants of the algorithm are

proposed by [86, 138]

2. **Coordinated Checkpointing Techniques**

   This technique involves the coordination of the processes in order to save their states. The coordination among processes maintains consistent global state of the system. Messages are used to maintain the coordination and this adds to the overhead of the method. Kumar and Hansdah [78] proposed a coordinated checkpointing technique that assumes nodes to be autonomous and do not block during checkpointing. The method can work efficiently with non-FIFO channels for which messages could be lost. In the method, any process can initiate a checkpoint by requesting permission from former coordinator. In essence, multiple coordinators are allowed; however, a single checkpoint is permitted at a given time.

   A coordinated multi-level diskless checkpointing is proposed in [56]. Stemming from the fact that as extreme scale systems are expected to be fully deployed and the probability of failure of these systems is high, saving checkpoints on disk posses I/O bandwidth disadvantages. Diskless checkpointing on the other hand suffers from redundancy problem. The authors then proposed an N-level diskless checkpointing approach that reduces the overhead that comes with tolerating simultaneous failure of less than N processors. This is achieved by arranging in layers, the diskless checkpointing strategies for a simultaneous failure of the processors. The algorithm follows four steps: The *first* is the determination of when and which level of checkpoint to execute; *secondly*, a consistent processor state is obtained by proper coordination; *thirdly*, the local memory checkpoints are obtained and lastly, these in memory checkpoints are encoded to particular checkpoint processes. The simulation results suggest that the method is promising and that the impact of inexact checkpoint schedules on the expected program execution is negligible especially if the checkpoint intervals are close to exact. Other similar but earlier methods are discussed

in [124] [136] [22]. Most of the methods highlighted in this section are also referred to as preventive checkpointing methods. One of the challenges with this method is that it is difficult to prevent a process from receiving application messages that could make the checkpoints inconsistent. Another problem is that computation is blocked during the checkpointing thereby causing delay.

3. **Hybrid Checkpointing Techniques**

   These techniques are a combination of two or more other checkpointing techniques. The approaches are mostly improvements on the other techniques or aimed at providing better checkpointing results.

   Bouguerra et al. [11] proposed a method that utilises the relationship between failure prediction and proactive checkpointing in combination with periodic checkpointing to reduce the effects of failure in the execution time of parallel applications. The rationale behind this is that the use of proactive checkpointing only is not sufficient enough to ensure a re-start of an application from the scratch. However, a combination of these methods can mitigate the effects and improves systems efficiency. In the approach, a prototyped state-of-the-art failure prediction, fast proactive checkpointing and periodic checkpointing methods are developed. Furthermore, the computing efficiency of the combined methods is captured using a mathematical model to obtain the optimal checkpointing interval. The method is evaluated by simulating the methods on large-scale supercomputer. Their result demonstrated that computing efficiency could be improved by as much as 30% using this method and mean time between failures (MTBF) is improved by a factor of two.

   Jangjaimon and Tzeng [71] developed an approach that aims at reducing the I/O bandwidth bottleneck to remote storage for checkpointing. The method is an adaptive incremental checkpointing that lowers this overhead by reducing the checkpoint files. This is done using delta com-

pression (where only the difference between current and previous file is written) combined with Markov model to predict points at which concurrent checkpointing can be done. Their method is reported to reduce applications turnaround time by about 40% when compare with static periodic checkpointing.

2.3.2 Task Migration

In this, a task is migrated to another node that is immediately available [11] when node failure is predicted to occur. Task migration eliminates cost associated with task restart in the face of node failure.

In a bid to avoid the disadvantages that arise with reactive fault tolerance, a method that complements the reactive approach to fault tolerance with proactive approach is proposed by Wang et al. [142] . Reactive fault tolerance often suffers from lack of scalability due to I/O requirements; hence, it relies on manual task resubmission whenever there is a node failure. The author's approach monitors the health information of nodes in order to perform earlier task migration whenever a node begins to show signs of health deterioration. They proposed a process-level live migration that supports continuous execution of the task without having to stop and restart again. The job migration process is described as follows: The health monitoring mechanism is equipped with sensors where different node properties are monitored. Such properties include: temperature data, fan speed and voltage. Upon deterioration of health of a node, it alerts an MPI-based decentralised scheduler. The scheduler chooses a replacement or destination node from the spare nodes. In the case of unavailable spare node, one with less workload is chosen. After a node is selected, migration of task is initiated; the memory snapshot of the process image is migrated to the destination node. The migration is done until the MPI tasks or processes are at globally consistent state. Finally, all communications with the MPI tasks are reconnected from the migrated processes. With experiments performed on a Linux cluster comprising of 17 compute nodes, the overheads associated with

the migration was assessed. The result shows that the overhead of live migration depends on the application and is higher than the execution time of normal rollback approach. Other similar approaches are seen in [111], [108].

In order to realise reduced system response time, Gupta et al. [52] proposed a method that adaptively schedules jobs and manages the resources based on external input. The jobs can be migrated to more processors and can also be shrunk based on the runtime conditions. Even though the approach focused on shrinking or expanding scheduler triggered jobs, it can be adopted to move jobs to safe nodes in the event of failure.

Munk et al. [98] recently proposed a concept that enable task-level migration in real time system many-core systems. It checks the viability of migrating a certain task at runtime. The migration is then performed on resources investigated to be available only. They performed migration through a 3-step procedure. The first is the decision phase, where, similar to [142], temperature monitoring and task runtimes profiling information is performed to decide if migration is necessary. In the case where it is necessary, a migration request is generated with set of possible destinations. The second is the investigation stage where the migration request is assessed. The possible destinations are evaluated for memory availability to store the task, determine if the task can be completed and when the migration task can best be performed. The third stage is the execution. It is triggered at the decision stage where the task on both the source and destination are activated.

The paper's idea is good, however, it lacks the empirical demonstration of the viability of the method.

2.4 Summary

In this chapter, we introduced a taxonomy of error detection and system recovery approaches for distributed computer systems. The chapter provided a comprehensive survey of the various methods in the area. The technique we

present in this thesis is the first to exploit the use of event logs combine with resource usage data for the task of error detection using an unsupervised approach.

Contributions of this chapter: The contributions of this chapter are the provision of comprehensive survey of error detection and system recovery techniques. To the best of our knowledge, it proposes the first taxonomy of the methods in the field of fault tolerance.

Most of the techniques discussed in this survey utilise systems data like error logs for performing detection. In the next chapter, we introduce the system and the data we use in this thesis.

Relation to other chapters: This chapter explains the previous studies upon which our approach is built. Hence it provides the fundamental literature to other chapters. The next chapter explains the systems and fault models upon which these techniques can be validated.

CHAPTER 3

System Description, Log Events And Fault Models

In this chapter, we detail the system model (Section 3.1) and the fault models (Section 3.2). We also explain the production systems (Section 3.3.2) we used and from which data is collected and we detail the structure of the log file data (Section 3.4). A summary of the chapter is presented in Section 3.5 including the chapter's contributions and relationship to other chapters.

3.1 System Model

A cluster system contains a set of interconnected nodes. These nodes run jobs that are allocated to them by a job scheduler. A node contains a set of production times during which scheduled jobs are executed. The cluster system also contains a set of software components (e.g. parallel file system) to support job execution. All the components involved writes log entries to a container. This is a typical model for most cluster systems like Cray, IBM Blue Gene/L, Ranger etc.

Specifically, we consider a cluster system CS to consist of the following set of entities: a set of K jobs, $J_1 \dots J_K$, a set of L nodes, $N_1 \dots N_L$, a set of M production time-bins $T1 \dots T_M$, a job scheduler JS and a set of software components C. The job scheduler JS allocates nodes (and communication paths among the nodes as required) and production times to each job. Each node in the cluster system maintains (monotonically increases) its own clock, and synchronization between the clocks of the nodes is assumed. Each job $J_i : 1 \leq i \leq K$, node $N_j : 1 \leq j \leq L$, job scheduler JS and any other software component may write messages to containers $U_1 \dots U_n$. Each node N_j and job J_i may transfer data to and from a file system FS. We assume such a

cluster system to be a black-box, i.e., access to software code is not permitted. However, we assume that access to the message logs which contain the failure events (written in containers $U_1 \ldots U_n$ by the jobs, nodes, job scheduler and any software component) is permitted.

3.2 Fault Model

We detail the fault models proposed as it relates to the systems investigated in this thesis.

3.2.1 Categories of Fault Model

In order to enhance and maintain system dependability, experts have been able to identify certain activities that could occur in a computer system and could potentially lead to failure. These identified activities are categorized to further assist in enhancing a system's fault tolerance. We explain some fault model categories and their relations to error detection.

Design-Runtime Fault Model

This model classifies faults based on their origin. Faults that emanate as a result of poor system design are regarded as *design faults*. *Runtime faults* on the other hand occur during systems production stage. These faults do not foster fault tolerance [6]. At the design stage, fault tolerance techniques are used to minimise or eliminate (if possible) these flaws using some engineering methods like system testing, formal specification etc. In situation where flaw-free system could not be achieved, runtime faults are inevitable. Error detection can detect such error patterns produced leading to the failure.

Permanent-Transient-Intermittent Fault Model

This model classification focuses on the duration of the faults. *Permanent faults* characteristically remain active so long as repairs are not made. These are

mostly damages to computer hardware. *Transient faults* occur temporarily and disappear. This fault may not occur again; an example may be a communication between node A and node B where a response to a request is not yet available at that time, but upon a resend of the request, it might be available. *Intermittent faults* are defects that occur due to system error and then disappear temporarily and could appear again. For example, when there is a loose connection in a system. This fault model is depicted in Figure 3.1, this structure is according to Siewiorek and Swarz [123] and elaborated in [115]. It can represent both hardware and software faults. For example, a software runtime fault can be a result of a wrong design. In relation to error detection, all these faults produces failure symptoms (error events); these event patterns or mis-behaviour can be detected to further enhance failure prediction or other failure analyses.

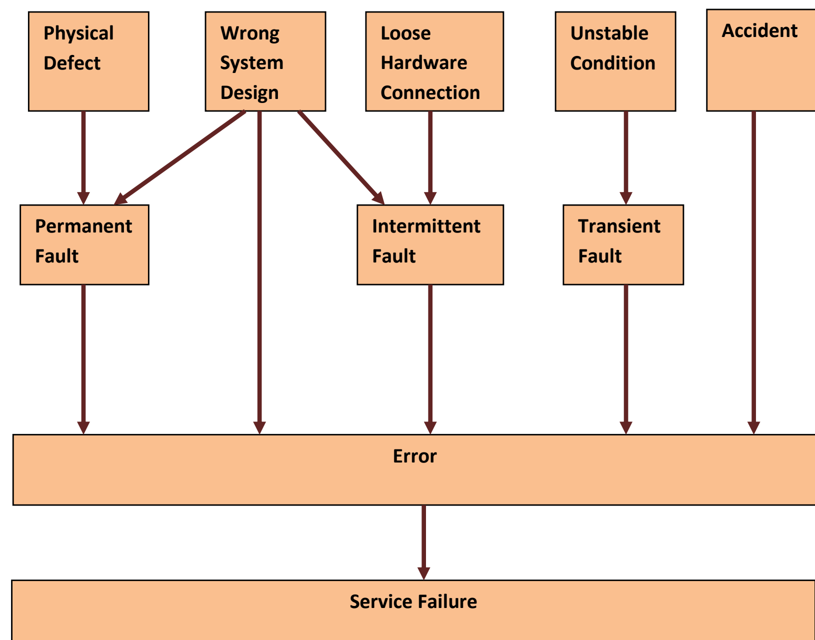


Figure 3.1: Permanent, Transient and Intermittent fault model

Hardware-Software-Human Fault Model

This model classifies faults based on the design and the operation of faults. In particular, the faults caused by hardware malfunction are called *hardware faults*, similarly, application-related faults are classified as *software faults*. Those that are the results of human activity are classified as *human faults*. Generally, all these faults like others explained before, produces error events or patterns that can depict such system misbehaviour. Error detection, which is a fault tolerance technique, can be applied to identify these patterns early enough to enable further proactive measures. In the case where error events are not produced, a fault may go undetected until a full failure is experienced since there would be no error patterns or signature for such failure.

Other Fault Models

Other fault models have been proposed; for example, the one by Gray and Reuter [48]. It is a model that focuses on software faults only. It classifies bugs based on how they are observed. Barborak et al. [8] characterises faults based on their behaviour or impact. He showed that some fault becomes difficult to detect at certain states, e.g. states where processing elements ceases to work.

3.2.2 Ranger and BlueGene/L Fault Models

As mentioned previously, a fault, when activated, could cause an error to exist in the system. The resulting state is then said to be erroneous. If the error is not detected and subsequently corrected, it can lead to a system failure [6]. When errors exist in the system, the system outputs what we call error messages, and the message part of the log entry captures the nature of the error in the system. The errors we focused on in this work are those that can lead to system failure [6], i.e., failure-inducing errors. A *failure-inducing* sequence consists of sequences of log messages that end in failure. A failure is typically characterized differently in different systems, for example, a failure in IBM BlueGene/L is characterized

by the *FAILURE* or *FATAL* severity level while, in the Ranger Supercomputer, the failures are mostly characterized by a *compute node soft lockups* messages.

A study of Ranger logs and the expert's knowledge has shown that soft lockups are some of the commonly occurring failures found in the Ranger supercomputer. Chuah et al. [23] has established that Machine Check Exceptions (MCE) and Evict/RPC error events signals likely occurrence of soft lockup failure in the cluster system. MCE is a way the computer's hardware reports about an error that it cannot correct. It will cause the CPU to interrupt the current program and call a special exception handler. Clusters' mean time to failure decreases with increase in nodes, hence making the need for error correction high. When the kernel logs an uncorrected hardware error, measures can be taken by the cluster software to rectify the problem by, for example, re-running the job on another node and/or reporting the failure to the administrator. Therefore, detection of MCE faults makes it possible to predict failures early. Soft lockup failure led by Evict/RPC Events are characterized by evict and recovery events. Chuah et al [24] have verified these hypotheses using a correlation and regression technique and obtained a high correlation between the MCE and Evict/RPC events and the soft lockup failure. Other fault events characterizing soft lockup failure include: memory access violation e.g. segmentation fault (segfault), network errors, internal interrupt conditions etc [25]. However, the sole occurrence of these faults may not necessarily lead to a failure, but rather a set of communicating components, each being affected by a fault can lead to system failure. All these faults falls within the categories of fault model explained earlier in Section 3.2.1.

3.3 Production Systems

Two production systems were studied in this research as case studies: the Ranger supercomputer and the Blue Gene/L systems.

3.3.1 Ranger Supercomputer

The Ranger supercomputer [60], the 15th ranked in the Top 500 supercomputer[1] list, consists of 4,048 nodes of which 3,936 are compute nodes and 78 are Lustre file system nodes. These nodes are connected via a high-speed Infiniband network. Each node generates its own log messages which are all sent to central logging system. Each node of a Ranger supercomputer runs a Linux Operating System kernel. Also, each node maintains its synchronization clock and the Sun Grid Engine powers its job scheduling process and resource management.

The Ranger supercomputer runs a Lustre file storage system. Lustre filesystem is an object-based high performance network file system that performs excellently for high throughput I/O tasks. It is a widely utilised file system in the supercomputing world. The file system is made up of:

- Meta-Data Server (MDS) that stores information like permissions, file names, directories etc. The MDS equally manages file requests from Lustre clients.
- Object Storage Server (OSS) which provides file I/O services. It also treats network requests from lustre clients.
- Lustre clients: The Lustre clients include visualization nodes, computational nodes, login nodes running the lustre paving way for file system monitoring.

The Ranger supercomputer runs TACC_stats [59], an I/O performance monitoring software. It monitors and records the resource usage/utilization by jobs on each node. The software runs on each node and the data collected on each of the nodes are logged centrally and synchronised.

3.3.2 The BlueGene/L (BGL) Supercomputer

The BlueGene/L deployed at Lawrence Livermore National Laboratory (LLNL), is currently ranked 12th in the Top 500 supercomputers list with a speed of *4.293*

[1] http://www.top500.org/lists/2015/06/

Pflop/s.

It is made up of 128K PowerPC 440 700MHz processors, that are arranged into 64 racks. Each rack is composed of 2 midplanes where the jobs are allocated on each. A midplane, with 1024 processors, contains sixteen node cards where the compute chips are contained, the input/output (I/O) cards that harbours the I/O chips and switches (24) where different midplanes connects.

3.4 System Data

Data from Ranger and BlueGene/L supercomputer systems are used in this research. This section explains the data in detail.

3.4.1 Ranger Event Logs

Most Linux-based cluster systems use the POSIX standard [1] for logging system events. This standard allows the freedom of formatting logs, which means that there are variations in different implementations. For example, different attributes are used by IBM's Blue Gene/L and Ranger to represent their components. Two different event logs are collected form the Ranger supercomputer system, they are: the syslog and the rationalized logs (ratlog).

syslog

An example of a Syslogs' *event* can be seen below:

Apr 4 15:58:38 mds5 kernel: LustreError: 138-a: work-MDT0000: A client on nid ..*.5@o2ib was evicted due to a lock blocking callback to .*.*.5@o2ib timed out: rc -107*

A Ranger syslog event (or log entry) has five attributes, namely: (i) **Timestamp** (*Apr 4 15:58:38*) containing the month, date, hour, minute and seconds at which the error event was logged. (ii) **Node Identifier** or **Node Id** (*mds5*) identifies the nodes from which the event is logged. (iii) **Protocol Identifier** (*kernel*) (iv) **Application** (*LustreError*) provides information about the source of the log entry, and (v) **Message** (*A client on nid *.*.*.5@o2ib was evicted due*

```
1:   Mar 29 10:00:44   i128-401  kernel:  [8965057.845375] LustreError: 11-0: an error
     occurred while communicating with *.*.*.36@o2ib. The ost_write operation failed with
     -122
2:   Mar 29 10:00:53  i128-401  kernel:   [8965077.319555] LustreError: 11-0: an error
     occurred while communicating with *.*.*.28@o2ib. The ost_write operation failed with
     -122
3:   Mar 29 11:27:16  i182-211  kernel:   [8981960.031578] a.out[867]: segfault at
     0000000000000000 rip 0000003351c5b2a6 rsp 00007fffdcd318c0 error 4
4:   Mar 29 11:27:16  i115-209  kernel:   [2073150.255467] a.out[22921]: segfault at
     0000000000000000 rip 0000003ad725b2a6 rsp 00007fffbf1a6d40 error 4
5:   Mar 30 10:02:24  i107-308  kernel:   [8966098.630066] BUG: Spurious soft lockup
     detected on CPU#8, pid:4242, uid:0, comm:ldlm_bl_22
6:   Mar 30 10:02:24  i107-308  kernel:   [8966098.642055] BUG: Spurious soft lockup
     detected on CPU#8, pid, uid:0, comm:ldlm_bl_22
7:   Mar 30 10:09:25  i107-111  kernel:   [8966563.203631] Machine check events logged
8:   Mar 30 10:09:51  i124-402  kernel:   [8965663.148499] Machine check events logged
9:   Mar 30 10:10:22  master    kernel:   LustreError:
     28400:0:(quota_ctl.c:288:client_quota_ctl()) ptlrpc_queue_wait failed, rc: -3
10:  Apr  1 05:23:54  i181-409  kernel:  [9203054.301173] Machine check events logged
11:  Apr  1 05:23:58  visbig  kernel:  EDAC k8 MC0: general bus error: participating
     processor(local node response), time-out(no timeout) memory transaction type(generic
     read), mem or  i/o(mem access), cache level generic)
```

Figure 3.2: Sample Log events for Ranger Supercomputer (syslog)

*to a lock blocking callback to *.*.*.5@o2ib timed out: rc -107*) provides more information regarding the system event and contains alphanumeric words and English-only words sequence. The English-only words sequence (*A client on nid was evicted due to a lock blocking callback to timed out*) is believed to give an insight into the error that has occurred. They are referred to as *Constant*. The alpha-numeric tokens (**.*.*.5@o2ib ,rc-107*) also called *Variable*, are believed to signify the interacting components within the cluster system. A detailed example of the Ranger logs is seen in Figure 3.2.

Rationalized logs

Rationalized logs [60] are similar to syslog, however, they contain an additional field which differentiates them from syslog. This additional information which is aimed at improving the effectiveness of log-based analysis and fault management, adds structure to the POSIX logs. A *job-id* field is an additional structure that contains a numeric number assigned to each running job. Other fields are similar to that of the syslog.

Message logs are mostly regarded as the only means and sources of information regarding the workings of a cluster system. It is used as the system

administrator's map to diagnosing faults in cluster systems. As the complexity of a system grows, so does the number of log entries, making the task of the system administrators increasingly complex to nearly impossible when faced with really large log files. The challenge with log data is that they are generally unstructured, often incomplete, with poor semantics and, most times, they have no particular message structure. Thus, we process our data by formatting it into a structure that is uniform and can give us the necessary information we need for our analysis. A careful investigation of the log entries showed that there is a pattern of occurrence of events before a failure.

A summary of the logs used in this research is seen in Table 3.1.

Table 3.1: Summary of Logs used from Production Systems

System	**Log Size**	**Messages**	**Start Date**	**End Date**
Ranger's syslog	5.6 GB	$> 2 \times 10^7$	2010-03-30	2010-08-30
Ranger's ratlog	4.3 GB	$> 2 \times 10^7$	2011-08-01	2012-01-30
	1.2 GB	$> 10^7$	2012-03-01	2012-03-30
Ranger Usage Data	52 GB		2012-03-01	2012-03-30
IBM's Blue Gene/L	730 MB	4,747,963	2005-06-03	2006-01-04

3.4.2 Ranger Resource Usage Data

Resource usage data are collected by TACC_stats [59] at Texas Advanced Computing Center (TACC). Basically, it is a job-oriented and logically structured version of the conventional *Sysstat* system performance monitor. TACC_stats records the hardware performance monitoring data, Lustre file-system operation counts and InfiniBand device usage. The resource usage data collector is executed on every node and is mostly executed both at the beginning and end of a job via the batch scheduler or periodically via cron. The collection of resource use data requires no cooperation from the job owner and requires minimal overhead.

Each stats file is self-explanatory and it contains a multi-line header, a schema descriptor and one or more record groups. Each stats file is identified by a header which contains the version of TACC_stats, the name of the

host and its uptime in seconds. An example of a stats file header is shown, for clarity:

```
$tacc_stats 1.0.2
$hostname i101-101.ranger.tacc.utexas.edu
$uname Linux x86_64 2.6.18-194.32.1.el5
 _TACC #18 SMP
Mon Mar 14 22:24:19 CDT 2011
$uptime 4753669
```

A schema descriptor for Lustre network usage parameters is seen below:

```
!lnet tx_msgs,E rx_msgs,E rx_msgs_dropped,
E tx_bytes,E,U=B rx_bytes E,U=B ...
lnet - 90604803 95213763 1068
808972316287 4589346402748 ...
```

A schema descriptor has the character ! followed by the type, and followed by a space separated list of elements or counters. Each counter consists of a key name such as *tx_msgs* which is followed by a comma-separated list of options. These options include: (1) E meaning that the counter is an *event counter*, (2) C signifying that the value is a control register and not a counter, (3) $W =< BITS >$ means that the counter is $< BITS >$ wide (32-bits or 64-bits), and (4) $U =< STR >$ signifying that the value is in units specified by $< STR >$ (e.g.: U=B where B stands for Bytes.). From the schema descriptor above, *lnet - 90604802* gives records of the number of messages transmitted in the Lustre network.

TACC_stats is open sourced and can be downloaded[2] and installed on Linux-based clusters. A list of the counters is shown in Table 3.2.

[2]https://github.com/TACC/tacc_stats

Table 3.2: List of 96 Elements of Resource Usage Data

Type	***Element***	***Quantity***
Lustre /work	read_bytes, write_bytes, direct_read, direct_write, dirty_pages_hits, dirty_pages_misses, ioctl, open, close, mmap, seek, fsync, setattr, truncate, flock, getattr, statfs,alloc node, setxattr, getxattr, listxattr, removexattr, inode_permission	23
Lustre /share	read_bytes, write_bytes, direct_read, direct_write, dirty_pages_hits, dirty_pages_misses, ioctl, open, close, mmap, seek, fsync, setattr, truncate, flock, getattr, statfs, alloc_node, setxattr, getxattr, listxattr, removexattr, inode_permission	23
Lustre /scratch	read_bytes, write_bytes, direct_read, direct_write, dirty_pages_hits, dirty_pages_misses, ioctl, open, close, mmap, seek, fsync, setattr, truncate, flock, getattr, statfs, alloc node, setxattr, getxattr, listxattr, removexattr, inode_permission	23
Lustre /network	tx_msgs, rx_msgs, rx_msgs_dropped, tx_bytes, rx_bytes, rx_bytes_dropped	6
Virtual memory	pgpgin, pgpgout, pswpin, pswpout, pgalloc_normal, pgfree, pgactivate, pgdeactivate, pgfault, pgmajfault_pgrefill_normal, pgsteal_normal, pgscan_kswapd_normal, pgscan_direct_normal, pginodesteal, slabs_scanned,kswapd_steal, kswapd_inodesteal, pageoutrun, allocstall_pgrotated	21

3.4.3 BlueGene/L Events logs

The IBM standard for Reliability, Availability, Serviceability (RAS) logs incorporates more attributes for specifying event types, severity of the events, job-id and the location of the event [88, 89]. The RAS events are logged through the Machine Monitoring and Control System (CMCS) and saved in a DB2 database engine. The time granularity for which events are logged is less than 1 millisecond. An example of IBM's Blue Gene/L (BGL) event is seen in Table 3.3.

Table 3.3: An example of event from Blue Gene/L RAS log

Rec ID	**Event type**	**Facility**	**Severity**	**Event Time**	**Location**	**Entry Data**
17838	RAS	KERNEL	INFO	2005-06-03-15 .42.50.363779	R02-M1-N0 -C:J12-U11	instruction cache parity error corrected

The RAS events recorded by IBM BlueGene/L CMCS each has the following attributes:

- *REC ID* is a sequential number given to each event (incrementally) as they are reported.
- *EVENT TYPE* specifies the logging method of the events which are more of reliability, availability and serviceability (RAS).
- *FACILITY* indicates the component where the event is flagged. This may be LINKCARD (problems with midplane switches), APP (flagged with applications), KERNEL(reported by the operating system), HARDWARE (facility related to system's hardware workings), DISCOVERY (relates initial configuration of the machine and resource discovery), CMCS, BGLMASTER, SERV NET (all reports events of the CMCS, network and BGLMASTER), and finally MONITOR facility (which are indicative of connection, temperature issues of the cards; they are mostly FAILURE events).
- *SEVERITY* can be one of these levels: INFO, WARNING, SEVERE, ERROR, FATAL or FAILURE in increasing order of severity. Regarding the reliability of the system, INFO events provide more information which are normal reports. WARNING events are usually related to dysfunctional cards. SEVERE events provide detailed information about the cards that are dysfunctional. ERROR events report persistent problems and their causes. FATAL and FAILURE events indicate more severe conditions that lead to application or software crashes. Unlike the last two, the first four severity levels are more of normal information and largely not severe.

- *EVENT TIME* is the time stamp for a particular event.
- *JOB ID* indicates the jobs that detects this event.
- *LOCATION* denotes where (nodes, cards) an error event occur.
- *ENTRY DATA* provides a description of the event reported.

3.4.4 Definition of Terms

In this section, basic definition of terms used regarding the event logs used is provided.

- **Event:** A single line of text containing various system fields such as $\langle timestamp, nodeID, protocol, application\rangle$ together with a log $\langle message\rangle$. The message reports the activity of the cluster system at time captured by $\langle timestamp\rangle$. Such an event is also often called a *log entry*. Whenever we refer to the message part of a log entry, we refer to this as the log message.
- **Event logs:** A sequence of events containing the activities that occur within a cluster system.
- **Similar events:** These are events with similar log messages, based on some notion of similarity. For example, from Figure 3.2, events 5 and 6 can be considered similar.
- **Identical events:** These are events believed to be exactly the same and/or are produced by the same "print" statement, e.g., events 7 and 8 in Figure 3.2.
- **Failure event:** This is an event that is often associated with and/or indicative of a system failure.
- **Event sequence:** A eventsequence consists of one or more events logged consecutively within a given time period, in order of increasing times-

tamp. In this paper, we use the terms *sequences*, *patterns* and *behaviours* interchangeably.

3.5 Summary

In this chapter, we explained the systems studied in this thesis. We provided details of the data used for error detection in this chapter; event logs and resource usage data from productions systems were also explained. We also detailed the fault models of the systems.

Relation to other chapters: The work in other chapters basically hinges on the fault, failures and the error events explained in this chapter. The methodologies proposed in subsequent chapters for error detection make use of the system and its data explained in this chapter for verification.

CHAPTER 4

Error Detection Using Clustering

Event logs are massive files, containing thousands of entries that are reported within small time intervals. The sheer number of entries makes proper analysis difficult. The difficulty of the task is compounded by the fact that a large number of events contained in the log contribute nothing to any meaningful analysis of the system. The need for preprocessing these logs for any meaningful and fast analysis cannot be over emphasized.

This chapter detailed our contribution towards preprocessing these logs to enhance detection and/or eventual prediction of failures. We explain a proposed failure sequence detection method in this chapter.

4.1 Introduction

Unscheduled downtime of large production computer systems, such as supercomputers or computer cluster systems, carries huge costs: (i) applications running on them have to be executed again, potentially requiring hours of re-execution, and (ii) a lot of effort is required to find and fix the causes of the downtime. These systems typically generate a lot of data, in the form of system logs, and these log files represent the main avenue by which system administrators gain insight into the behaviour of such computer systems [104]. To enhance the availability of such large distributed systems, the ability to detect errors[1] is important. However, this is a challenging task, given the size and scale of such systems. Typically system administrators will resort to accessing log files for error detection.

[1] An error is the symptom of a fault in the system [6].

4.1.1 Log Size and Structure

However, due to the *size* of such data files and the complexity of such systems, system administrators usually adopt a divide and conquer approach to analysing the data. An individual line of the system log may not impart much information to a system administrator due to the lack of context to sufficiently characterize a message. At the other end of the spectrum, huge log dumps with interleaved node outputs make it difficult for system administrators to capture the communication relations among nodes. Researchers have adopted the notion of *node hours*, which represents all the messages a given node generates in a given hour [106], to achieve a tractable structure. In any case, a line in the log file will either be *normal*, capturing a normal event occurring in the system or *faulty*, capturing erroneous behaviour in the system.

Further, recovery in such systems, e.g., checkpointing, are typically computationally expensive, requiring that these steps are taken if there is a possible impending failure of the system.

4.1.2 Errors and Failures

Typically, only a proportion of event logs that are errors will lead to system failures. As such, system designers wish to focus on such errors, which we term as *failure inducing* errors[2]. The presence of an error in the system can be inferred when faulty events can be observed: failure prediction allows system administrators to take remedial steps earlier, e.g. by rebooting a node [103]. Several approaches have addressed the problem of error or alert detection such as [103, 106] or finding the presence of errors in message logs [129]. These approaches use labelled node hours in conjunction with supervised learning to predict whether a given node hour contains an alert or fault. Labelling of the data is usually done manually by experts, which is a very expensive process, requiring highly specialist knowledge. Therefore, such analysis techniques are

[2]Henceforth, whenever we say error detection, we mean failure-inducing error detection. Since the aim is to detect failure-inducing errors.

invariably expensive due to the size and nature of the log files being processed.

4.1.3 Event Logs Redundancy

The event logs are unstructured and highly redundant, i.e. several lines can be related to a single event in the system as earlier explained above.

To address the redundancy problem, the logs are typically *filtered*, or *pre-processed*, to retain only those events that are most relevant to the log analysis. Since these log files are highly redundant, a high filtering rate will significantly reduce the size of the file, thereby reducing the computation time of the log analysis process. The *filtering* or *compression* techniques (we use both interchangeably), are used to remove events that are not deemed useful for analysis. A common problem with such compression techniques is that they may remove important information that is pertinent to the analysis phase, as captured by the targeted high compression or filtering rate [154].

Overall, typical log analysis techniques currently apply a filtering phase to remove redundancy in the labelled log files. The compressed annotated log files are then analyzed according to system requirements.

4.1.4 Objectives of the Chapter

This chapter of the thesis seek to achieve the following objectives:

- Propose a method for preprocessing the large logs to reduce it to a manageable size without filtering out important events. Particularly, the chapter seek to detail a *novel and generic* approach to log filtering that not only filters redundant events but also preserve events that are not similar but causality related. This is to preserve event patterns that serve as precursor to failures. At this filtering stage, *timestamps* and *node ids* of the events becomes crucial.

- Propose an *unsupervised* approach that eschew *fault labeling* (as it is computationally expensive to label the data) to detect failure-inducing or er-

roneous patterns among the unlabeled log message sequences that lead to system failures.

This chapter is structured as follows: In Section 4.2 the problem addressed in the chapter is outlined. Sections 4.3, 4.4 and 4.5 respectively include our methodology for filtering the event logs, transforming the logs into matrix form and failure pattern detection. In Sections 4.6 and 4.7, we respectively explain the experimentation methods and discuss the results obtained when applying our methodology to log data from production systems. We summarise the chapter's work in Section 4.8.

4.2 Problem Statement and Methodology Overview

A cluster log contains an interleaving of normal and error messages. When there is no error in the system, only sequences of normal messages are output. However, when one or more jobs are affected by errors, these components output error messages, that are logged. As previously argued, not all (combination of) errors lead to system failures. Therefore, it becomes important to identify such sequences that are likely to result in system failures.

Observation of such sequences may then become a precursor to an impending failure, upon which recovery mechanisms can be built. However, the size and complexity of such systems, the nature of the log messages (varying according to the operating system, networks, file systems), the timing and frequency of occurrences of these error messages make it difficult to accurately capture behaviours which are precursors to system failures. Specifically, error detection is difficult as there are several interactions among system attributes that one would need to consider to accurately capture failure-inducing behaviours.

We now specify three requirements for a log analysis methodology for error detection: the methodology needs to (i) have a low computational overhead and (ii) have a high accuracy for error detection, while minimising the number of false positives.

For the first requirement, our approach reduces the number of unique events under consideration by grouping similar ones together, assigning these events a unique id, marking them as the same event and using this information to prune the log data. For the second requirement, we eschew labelling the data *with error or failure information*, due to the high overhead associated with such techniques[3]. We thus focus on *unsupervised* learning methods, which constitute a novel approach to this problem.

Thus, the problem we tackle in this chapter is the following: Given an error log file E that consists of a sequence of messages $m_1 \ldots m_N$, where N is the total number of lines in the log file, a time window τ over the message sequences such that there is no more than a single failure during τ, we seek to develop a methodology that identifies failure-inducing patterns. Some of the challenges are: (i) similar message sequences may end up with different outcomes as there is a successful recovery in one of the sequence, (ii) two sequences may be similar in terms of the set of messages output but follow different temporal patterns.An overview of the steps taken to solve these challenges is shown in Figure 4.1.

The methodology consists of 3 main steps: (i) the first step is to transform the log data into a suitable format for data analysis, (ii) the second step purges redundant log messages to enable efficient analysis and (iii) the final step detects failure-inducing patterns through an unsupervised learning approach. An example of redundant event logs is seen in the sample event sequence shown in Figure 3.2. It contain messages (e.g., log events 1 and 2 or log events 7 and 8) that can be classified as redundant. Such redundant messages only distort failure patterns [117]. Hence, logs need to be preprocessed to remove the redundant events while keeping the relevant ones. Finally, we apply an unsupervised learning technique can be used to detect errors in the log files. We detail each step in the next few sections.

[3]Observe that we do label the data with id information for the first requirement. We label the data only to keep the runtime low and do not add labels that require expert knowledge.

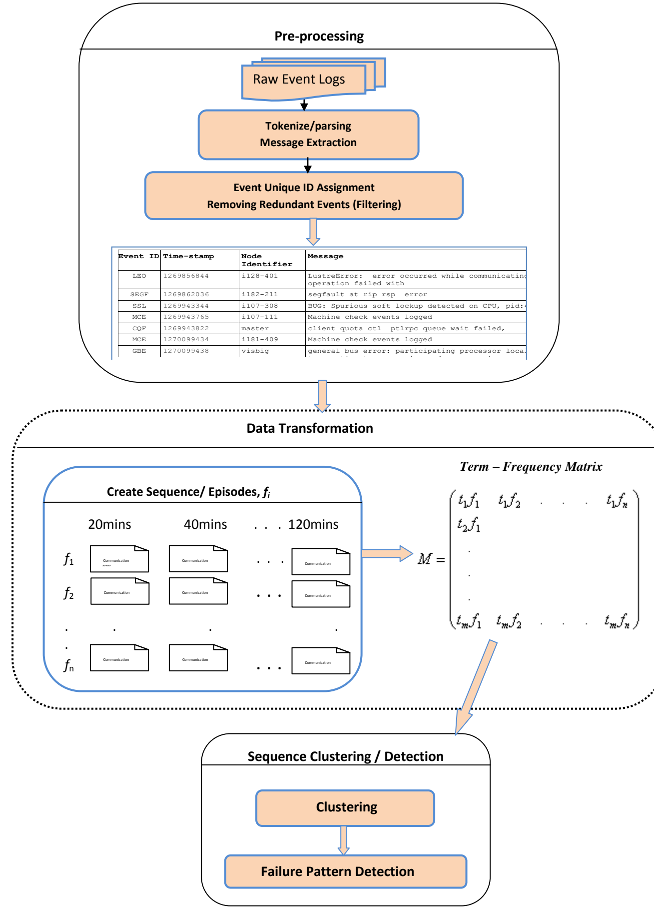


Figure 4.1: Methodology Work flow showing the steps taken to achieve the objectives

4.3 Preprocessing

This section detail our approach for filtering redundant logs and the normal approaches usually done for filtering. It first explains the general preprocessing steps done in log analysis.

4.3.1 Log Events Preprocessing

We detail the steps that are most important in processing the log files into a format suitable for analysis.

Tokenization and Parsing

This phase involves parsing the logs to obtain the event types and event attributes, using simple rules. Tokens that carry no useful information for analysis are removed. For example, numeric-only tokens are removed but *attributes* (alpha-numeric tokens) and the *message types* (English-like only terms) are kept. Also, fields like protocol identifier and application are removed or omitted during the parsing and tokenizing phase.

The message part contains English words, numeric and alphanumeric tokens. The English tokens provide information pertaining to the state of the system and contribute to meaningful patterns. The alphanumeric tokens capture the interacting components or software functions involved. These interacting components, which do not occur frequently and show less or no pattern, are also important since we are interested in interacting nodes of the cluster system. On the other hand, the numeric only tokens are removed as they only add noise.

Time Conversion

An event in a cluster system is logged with the time at which the event occurred. The timestamps are very useful for any meaningful log analysis or time series analysis. However, the time format reported is not readily suitable for manipulation purposes, e.g. matrix manipulation. In the Ranger logs, for example, a reported timestamp *2010 Mar 31 15:56:57* is for an event that occurred at 15th hour, 56th minute and 57th second on March 31, 2010. We then convert this to the epoch timestamp format which gives a value of *1270051017* for the above timestamp, which can then be easily manipulated. The Unix epoch (or Unix time or POSIX time or Unix timestamp) is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT)

4.3.2 Log Compression: Removing Redundant Events

This section explains our log compression or filtering approach, where redundant log events are removed. In the first step, we process the events such that similar events are assigned the same unique id. Then, we remove those ids that are not relevant in the sequence.

Obtaining Unique Events

One way of developing a technique with low runtime is to reduce the amount of data to be analyzed, whilst keeping the relevant ones. Our proposed way of achieving this is to develop an approach whereby events that are very similar to each other or deemed identical are assigned the same id and, hence, redundant events are then purged from the log file.

There are sets of events whose messages may be syntactically different (resp. similar) but semantically similar (resp. different), making it challenging to accurately capture the similarity of messages. However, the intuition that is used here to support the identification of similar messages (with similar meaning) is as follows: *if two sentences (i.e. word sequences) are very closely related in the order of the words, then it is very likely that they will have very similar meaning.* Thus, to identify similar events, we first extract the log messages of the events, we then define an appropriate distance metric between two messages to capture their similarity based on the word sequence. We also require the notion of *distance threshold* to bound the distance between two messages that are considered similar. We use the *Levenshtein distance* as a metric of sequence similarity.

Levenshtein Distance (or Edit distance): We make use of *Levenshtein Distance* (LD) [85] metric to capture the between two messages, LD is equally defined on pairs of strings based on edit operations (i.e., insertion, deletion or substitutions) on the characters of the strings. Hence, the Levenshtein distance between two strings s_1 and s_2 is the number of operations required to transform s_2 into s_1 or vice versa, as it is symmetric, assuming all operations count the

same. LD is an effective and widely used string comparison approach. Since we have messages, we extend LD to words and define it as follows: It is defined based on edit operations (i.e. insertion, deletion or substitutions) on the words of the messages. LD is found to be more suitable here than cosine similarity metric, since the latter is a vector-based similarity measure.

From the sample logs of Figure 3.2, it can be observed that it is necessary for any similarity metric used to consider the order of the terms in the log messages to obtain a meaningful result. For example, the log messages *…error occurred while communicating with…* and *…Communication error occurred on…* may appear similar but may be semantically different. A similarity metric that does not take order of terms into consideration (i.e., a metric considering a sentence as a set of words) will cluster these events together, i.e., these events will be seen as similar, because they have similar terms. To address this challenge, we use the Levenshtein distance on terms, without transposition, taking term order into consideration. Also, defining this metric based on terms reduces the computational cost incurred, as opposed to when it is defined on string characters.

We thus propose an algorithm (see Algorithm 1) that uses LD to first find the similarity between log messages and then group similar events based on similarity value. Then, events in the same cluster are indexed with the same id.

Finally, to bound the similarity of events, we define a *similarity threshold*, with the lesser the number of edits, the higher the similarity. Hence, we define the threshold such that, when the edit distance between a pair of messages is less than or equal to the threshold, λ, (hence highly similar), these events are regarded as similar and hence clustered together.

Event Similarity Threshold: As argued previously, two different messages are likely to have a similar meaning if they differ in very few places only. Using an *iterative approach* [61], we start with a small value of similarity threshold, λ, then increase the value in small increments and monitor the output, until a satisfactory similarity value is obtained using the log data. At this point,

we observed that similar events are indeed grouped together. For a very small similarity threshold (i.e., 0 or 1), only messages that are exactly similar are clustered together. At the other end of the spectrum, a high value of λ will result in messages with different meanings (and structure) to be grouped as similar. After using an incremental technique, a threshold of 2 was chosen, i.e, $\lambda = 2$.

We now propose an algorithm (Algorithm 1) that groups together similar messages, according to the LD distance, and labels them with the same id. The algorithm proceeds as follows: In the first step, it groups together messages with the same number of tokens, i.e., messages of the same length. This is because event messages with same token length are more likely to have been produced by same *printf* statement or from same node and reporting similar happening. Another reason is that we are not looking at the semantic meaning of the event messages (which will involve natural language processing techniques), but the events that similar in terms what produces them. Subsequently, the following step goes through each group and partitions them into smaller groups based on LD. An example of the output of Algorithm 1 is shown in Figure 4.2.

Algorithm 1 An algorithm for grouping similar events

```
 1: procedure GroupSimilarEvents(log events e_1, ..., e_n, Min Similarity Threshold, λ)
 2:     for all log events e_i, i = 1 ... n do
 3:         group events based on their token length
 4:     end for
 5:     for each group g do
 6:         for each pair of message (m_i, m_j) in g do
 7:             if LD(m_i, m_j) <= λ then
 8:                 group m_i and m_j together
 9:             end if
10:         end for
11:     end for
12:     assign a unique id to each group
13:     Return() {outputs log events with their cluster ID}
14: end procedure
```

These logs still contain redundant events, for example, events 5 & 6 (please observe that events 5 and 6 are clustered together, though being slightly dif-

	Event ID	Time-stamp	Node Identifier	Message
1	LEO	1269856844	i128-401	LustreError: error occurred while communicating with 129.114.97.36@o2ib. The ost_write operation failed with
2	LEO	1269856853	i128-401	LustreError: error occurred while communicating with 129.114.97.36@o2ib. The ost_write operation failed with
3	SEGF	1269862036	i182-211	segfault at rip rsp error
4	SEGF	1269862036	i115-209	segfault at rip rsp error
5	SSL	1269943344	i107-308	BUG: Spurious soft lockup detected on CPU, pid:4242, uid:0, comm:ldlm_bl_22
6	SSL	1269943344	i107-308	BUG: soft lockup detected on CPU, pid:21851, uid:0, comm:ldlm_bl_13
7	MCE	1269943765	i107-111	Machine check events logged
8	MCE	1269943791	i124-402	Machine check events logged
9	CQF	1269943822	master	client quota ctl ptlrpc queue wait failed,
10	MCE	1270099434	i181-409	Machine check events logged
11	GBE	1270099438	visbig	general bus error: participating processor local node response, time-out no timeout memory transaction type generic read, mem or io mem access cache level generic

Figure 4.2: Sample pre-processed logs of Figure 3.2

ferent, and are indexed using the same id). This necessitated next section (removing redundant events).

Removing Redundant Events

According to Iyer and Rosetti [68], the occurrence of similar or identical events within a small time window is likely caused by the same error, thus, these messages are potentially related as they could point to the same root-cause. This means that not all of these events are needed during analysis since they are redundant. Therefore, removing these redundant messages may prove to be beneficial to the analysis stage. In filtering of redundant log events, we consider events in a sequence having the following properties:

- Similar events that are reported in sequence by the same node within a small time window are redundant. This is because nodes can log several similar messages that are triggered by the same fault, e.g., events 5 and 6 of Figure 4.2.
- Similar events that are reported by different nodes in a sequence and within a defined time window are redundant. This could be triggered by

the same fault resulting in similar mis-behaviour by those affected cluster nodes.

- Identical events occurring in sequence (consecutively) and within a defined small time window are redundant.

Normal filtering will keep the first event in a sequence of similar events and remove the rest [154]. It is pertinent to note that it is also possible for error messages logged by different nodes within close time intervals to be caused by different faults, while some events are causally-related (emanate as a result of the same fault). In this work we do not discard such events. The process of identifying and grouping the error events exhibiting the above properties is done using a combination of both tupling and time grouping heuristics [61]. We define some heuristics that capture the properties outlined above.

Normal Filtering

Filtering or compression is a process used to reduce the complexities associated with log analysis. It is generally agreed that filtering or pre-processing logs is an important process [88, 154]. The process eliminates redundant events from logs, thereby reducing the initial huge size of the logs. This however, must avoid the purging of useful events or event patterns that are important for failure pattern detection. In normal log filtering [88], events that repeats within certain time window are removed, only the first event is kept. This simple log filtering we refer to here as *normal filtering*. Normal filtering however, can remove fault events that are relevant for analyzing causal correlations among events. On the other hand, Zheng et al. [154] proposed an approach that can filter causally-related events or what they termed as semantic redundancy. Their approach can preserve useful patterns and has been shown to be better than normal filtering. However, their approach is log-specific. It is dependent on the fact that logs must be labelled with severity information. The approach *would not work* on logs without severity information and hence cannot be generalized as most logs are not pre-labelled with severity information. As a matter of contrast, we propose

an approach that workS on any log with or without severity information.

Job-id based Filtering

With careful observation of the logs and through experts' input, we realised that achieving a high compression rate while preserving patterns is important and dependent on how informative a given log is. For example, Ranger's *Ratlogs* events are labelled with more information regarding the nodes and jobs involved, providing a richer description of an event. The *Job-id* field in logs indicates a particular job that detects the reported event. The job-ids, when correlated with failure events, tell us which jobs are the likely source of the failure. This implies that identical job-ids present in different events within the same sequence would likely have high correlation with the failure that is eventually experienced [24]. In order to achieve high *event compression accuracy* (ability to keep unique events) and *completeness* (remove redundant events), we propose a filtering approach that removes redundant events or events that are related based on sources, similarity and time of occurrence.

Specifically, given two events e_1 and e_2, with times of occurrence Te_1 and Te_2 respectively, these are causally-related or emanate as a result of the same faults, and are hence redundant, if:

- $nodeid(e_1) == nodeid(e_2)$ && $jobid(e_1) == jobid(e_2)$ && $|Te_1 - Te_2| \leq tw$ && $sim(e_1, e_2) \geq \lambda$,

where $sim(.)$ is the similarity given by LD, λ is the similarity threshold, tw is the time window for which events e_1 and e_2 are similar and can be filtered. Table 4.3 depicts a sample filtered logs with redundant events removed.

4.4 Data Transformation

This involves translating the processed (filtered) data into a format that can capture sequences of events into a matrix that can be readily used by any analysis algorithm.

Event ID	Time-stamp	Node Identifier	Message
LEO	1269856844	i128-401	LustreError: error occurred while communicating with 129.114.97.36@o2ib. The ost_write operation failed with
SEGF	1269862036	i182-211	segfault at rip rsp error
SSL	1269943344	i107-308	BUG: Spurious soft lockup detected on CPU, pid:4242, uid:0, comm:ldlm_bl_22
MCE	1269943765	i107-111	Machine check events logged
CQF	1269943822	master	client quota ctl ptlrpc queue wait failed,
MCE	1270099434	i181-409	Machine check events logged
GBE	1270099438	visbig	general bus error: participating processor local node response, time-out no timeout memory transaction type generic read, mem or io mem access cache level generic

Figure 4.3: Sample preprocessed event logs (syslog) with redundant event removed

Event Sequences

Consider the sequence of events $e_1, e_2, \ldots, e_n$ in Figure 4.4. Assuming that events e_k, e_{k+j} are failure events; we define the *time window, tw* for failure e_{k+j} as the period between events at times, say, $\ell - \tau$ and ℓ, where the event at time ℓ is the failure event (e_{k+j}) under consideration and time $\ell - \tau < \tau < \ell$, is time of an event which occurs before the failure event at time ℓ, which is event e_3. The sets of events within this period make the *failure episode* [24], where an *event sequence* or *pattern* can be a subset of the failure pattern. So failure sequence F_j includes all events up to that point, at time ℓ.

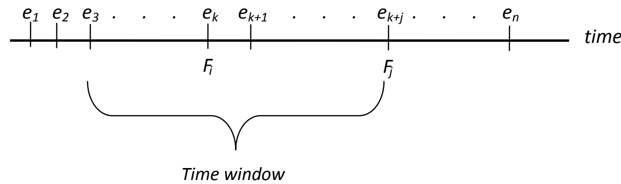


Figure 4.4: Event logs sequence

Feature Creation

Events within sequences are believed to generate empirical probability distribution implying relative probability of occurrence of each event within a se-

$$K = \begin{pmatrix} F_1t_1 & F_2t_1 & . & . & . & F_Nt_1 \\ F_1t_2 & & & & & \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ F_1t_M & F_2t_M & . & . & . & F_Nt_M \end{pmatrix}$$

Figure 4.5: Data matrix K of N sequences, where F_jt_i is the number of counts of message term t_i in sequence F_j.

quence [26]. In other words, the frequency of message types captures events distribution or observation and its changes within patterns. This informed our decision to utilise the message term-frequency transformation. More specifically, the messages from each sequence are transformed into a term - frequency matrix (see Figure 4.5). The rows of the matrix represents the different terms while the columns are the sequences. A failure sequence consists of events that precede the failure event within a given time window.

Hence, given N sequences extracted within a time window, then the matrix K, for this will contain M rows of message terms and N columns of sequences, as shown in the matrix of Figure 4.5.

Matrix Normalization: The term weights or cumulative frequency across sequences are normalized to a value within 0 and 1. This enables easy handling and interpretation of the data and reduce sensitivity towards high variance data. The frequencies are normalised by the formula in Equation 4.1.

$$n_t(i) = \frac{w_t(F_i)}{\sum_{j=1}^{N} w_t(F_j)} \tag{4.1}$$

where n_t is the normalized message term, $w_t(F_i)$ is the count of message term t of sequence F_i. It is divided by the sum of the count of term t across all sequences; hence, each column forms a *probability distribution.* For the sake of simplicity, we represent these column vectors (or sequences) as $F_1, ..., F_N$ subsequently in our clustering algorithm and the columns are the inputs.

4.5 Sequence Clustering and Detection

4.5.1 Clustering

Given a set $\mathbb{K} = [F_i \ldots F_N]$, of N input entries (columns), where $F_i = \{F_i t_1, F_i t_2, ..., F_i t_M\}$, then each input is either a normal event pattern or faulty one. Groups of sequences or patterns that exhibit similar characteristics due to the presence of similar faults generating similar message types within a sequence or pattern can be identified through clustering. Hence, in order to group such similar sequences we employ distance based clustering.

For the purpose of faulty sequence detection in event logs, it is important to define a distance metric that could capture the informativeness of the sequence and/or correlation patterns between the events. Two metrics are defined as explained below.

- *Jenson-Shannon Divergence (JSD)* metric measures the divergence or similarity between two or more probability distributions. Events of log data are sometimes infrequent and spatial in distribution (can occur randomly in different sequences). JSD has been shown to be effective in capturing relationships among tokens of such distribution [97], hence we utilised it in this work. Considering two distributions F_i, F_j (*note that we have established earlier, that the input vectors are probability distributions*), JSD shows how much information is lost when using one of F_i or F_j to approximate the other.

  Hence, given two sequences (in this case, the input column vectors), F_i, F_j, then JSD is defined by

$$JSD(F_i, F_j) = \frac{1}{2} KLD(F_i \| E) + \frac{1}{2} KLD(F_j \| E) \tag{4.2}$$

  where KLD is the Kullback Divergence [34] given as $KLD(F_i \| F_j) = \sum_{k=1}^{M} t_k F_i \ log(\frac{t_k F_i}{t_k F_j})$ and $E = \frac{F_i + F_j}{2}$

Hence, the similarity between F_i and F_j is given by:

$$sim(F_i, F_j) = |1 - JSD(F_i, F_j)|. \tag{4.3}$$

The values range between 0 and 1, with values closer to 0 implying more dissimilar sequences and values close to 1 implying similar type of sequences.

- The second metric, *Correlation Metric* (Corr), is based on the correlation between sequences. Given any two columns from matrix K, the correlation distance between them is given below, where *cov* is the covariance and *std* is the standard deviation.

$$\text{cov}(F_i, F_j) = \frac{1}{M}\sum_{k=1}^{M}(F_{i,k} - \overline{F_i})(F_{j,k} - \overline{F_j}) \tag{4.4}$$

$$std(F_i) = \sqrt{\frac{1}{M}\sum_{k=1}^{M}(F_{i,k} - \overline{F_i})^2} \tag{4.5}$$

where $\overline{F_i} = \frac{1}{M}\sum_{k=1}^{M} F_{i,k}$

$$sim(F_i, F_j) = 1 - \left|\frac{\text{cov}(F_i, F_j)}{std(F_i) * std(F_j)}\right| \tag{4.6}$$

where $F_{i,k} = t_k F_i$ is the value of the frequency of message term t_k in sequence F_i. We treat $sim(.)$ as similarity or distance measure between two feature vectors. Two clustering algorithms are proposed as explained below and seen in Algorithms 2 and 3.

Naïve Clustering Algorithm

Different faults may induce similar error manifestation in the system. Therefore, in this algorithm, the basic assumption is that similar sequences are likely to have been generated by the same fault type; therefore, all data points (se-

quences) which are close enough based on a similarity metric are clustered together (Algorithm 2). The algorithm first initialised each data point (sequences) as clusters such that all the sequences form sets of clusters. In that case, a copy of the the set of clusters C is created. All the clusters with high similarity values, sm (greater or equal to threshold), are considered to contain similar pattern and hence group together. However, those patterns different from all others are keep alone as singleton clusters. The result of this algorithm is a set of clusters of similar patterns.

Algorithm 2 Naïve clustering of event sequences

```
1: procedure NaïveClustering(event sequences 𝕂 = F_1, F_2, ..., F_N, SimilarityThreshold, δ)
2:     Initialize each F_i as a cluster of its own, all belonging to cluster set C; Set of clusters ℂ is the output.
3:     for each F_i ∈ 𝕂 do
4:         for each cluster c_k ∈ C do
5:             for all members, V_j ∈ c_k do
6:                 sm = sim(F_i, V_j);
7:                 if sm >= δ then
8:                     add F_i to cluster c_k
9:                 else
10:                    if (last cluster) then
11:                        create a new cluster, c = F_i
12:                        add c to ℂ
13:                    end if
14:                end if
15:                if V_j = last cluster point of c_k then
16:                    add c_k to ℂ
17:                end if
18:            end for
19:        end for
20:    end for
21:    Repeat step 3 Until all sequences are clustered
22:    Return() {outputs ℂ, clusters containing event sequences}
23: end procedure
```

Hierarchical Agglomerative Clustering Variant

The motivation for this algorithm stems from the fact that time at which a fault is experience may affect the nature of the events despite the fact that the sequences may have similar faults and secondly, sequences have a high tendency

of belonging to more than one cluster, that is, patterns can be similar due to the fact that similar computers executing similar jobs are likely to produce similar messages [106]. In order to obtain more cohesive clusters of sequences, we introduce a variant on hierarchical clustering, which we refer to as HAC. The HAC algorithm (Algorithm 3) targets those sequences we refer to as *borderline sequences* (sequences with the possibility to belong to more than one cluster), so they are clustered in the right group in order to obtain the different sequence characteristics for detection. These are sequences with the tendency to belong to more than a cluster due to the presence of similar error messages. In HAC, such "borderline" sequences are captured as sub-clusters of cluster with higher closeness value. A cluster SC is a sub-cluster of cluster C, if $|SC| < |C|$, and if the *validity index* of C, ($val_ind(C)$), is greater than that of SC and the similarity between their *centroids* is greater than or equal to the threshold, δ. The *Validity index* [58], referred to in our algorithm as ($val_ind(C)$), is a measure of goodness of cluster C by finding the compactness or how close elements of the cluster are and how separate it is from other clusters. We calculated this using the Silhouette coefficient [58], given by equation 4.7.

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)} \tag{4.7}$$

where s_i = silhouette coefficient for sequence i, b_i = minimum(*average distances of sequence i with sequences of other clusters*), a_i = average distance of i to sequences in its own cluster. Hence, the goodness of the cluster is the average of all the silhouettes, s_i, of the clusters; and $-1 \leq s_i \leq 1$, with value of s_i close to 1 indicating that the sequences clustered together are similar and values closer to -1 indicating less similar sequences.

4.5.2 Detection of Failure Patterns

The aim of clustering is to group similar sequences or patterns together, which are either normal or failure-inducing. It is hypothesized that similar sequences

Algorithm 3 Hierarchical Agglomerative Clustering (HAC) algorithm for event sequences

1: **procedure** HAC($c_i \in \mathbb{C}$, clusters of event sequences, SimilarityThreshold, δ)
2: SC= sub-clusters
3: **Sort** $\mathbb{C}$ according to cluster size
4: **for** $i \leftarrow 0$ to $|\mathbb{C}| - 1$ **do**
5: **for** $j \leftarrow i + 1$ to $|\mathbb{C}|$ **do**
6: Find similarity, sim, of **centroids** of clusters c_i and c_j;
7: **if** $sim >= \delta$ **then**
8: **if** $val_ind(c_j) > val_ind(c_i)$ **then**
9: add c_j to set of potential sub-clusters, s_i of c_i.
10: **end if**
11: **end if**
12: **end for**
13: **for** all potential sub-cluster s_i **do**
14: add s_i to SC if not previously a sub cluster or if its sim is greater than its previous sim value.
15: **end for**
16: **end for**
17: Return(SC) {}
18: **end procedure**

leading to failure will be clustered together and that the same holds for good sequences. Event log sequences can be classified as noisy, periodic or silent in their behaviour [43]. Noisy sequences occur with high frequency (bursty or chatty) and the level of interaction of the nodes involved increases within short period. The high level of interaction may depend on the type of faults that has occurred. For example, *communication errors* generally result in spurious event reporting by the nodes involved. Gainaru *et al.* [43] have shown that such events are often symptoms that precede a failure. Normal sequences, on the other hand, are mostly periodic. In this case, the sequences are characterised by normal cluster status reports and moderately low interaction of components over a longer time window.

In our approach, we harness these properties in order to characterise failure sequences. We consider the measure of information of the sequences in clusters and also how informative the message terms of the sequences are. These scenarios can be captured by the *Mutual Information* (I) of sequences and *Entropy*

(H) of message types. Both I and H are concepts that captures the informativeness of a given random variable. In our context,I captures common behaviour of the interacting nodes within a sequence while H on the other hand, captures the uncertainty or unpredictability of the event types within a sequence. To further support our approach, it has been argued that *changes observed in entropy are good indicators of changes in the behaviour of distributed systems and networks* [79]. Our idea is motivated by Brown's clustering algorithm [13], Percy Liang's work on terms cluster characteristics and quality [88] and log entropy [10]. We hypothesize that sequences with higher uncertainty (entropy) and reduced I signifies abnormal system behaviour and failure sequences with the converse true for normal systems behaviour.

Hence, given a set C of m clusters $c_1, ..., c_m$ and $c_k = \{F_1, ..., F_N\}$, containing set of similar event sequences, then, Mutual Information $I(c_k)$ is given as:

$$I(c_k) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} p(F_i, F_j) \log \frac{p(F_i, F_j)}{p(F_i)p(F_j)} \tag{4.8}$$

where $p(F_i, F_j)$ is the joint probability distribution of sequences of cluster c, $p(F)$ is the probability distribution of sequence F. Similarly, the *Entropy (H)*, is given as:

$$H(c_k) = -\sum_{i=1}^{N} \sum_{j=1}^{M} F_i t_j \log F_i t_j \tag{4.9}$$

where $F_i t_j$ is the distribution of the terms t_j of sequences F_i in a cluster c_k.

We obtain the informativeness of a cluster c as:

$$\varphi(c) = I(c) - H(c) \tag{4.10}$$

Hence, from Equations 5.4,5.5 and 4.10, detection is achieved as follows:

$$f(c) = \begin{cases} 1 & \text{if } \varphi(c) < 0 \\ else & \begin{cases} 1 & \text{if } \varphi(c) > \tau \ \& \ H(c) > 0 \\ 0 & \text{otherwise} \end{cases} \end{cases} \tag{4.11}$$

cluster $f(c)$ is detected as containing *failure sequences* if its value of informativeness is high or not(depending on threshold) as shown in equation 4.11. Where τ is detection threshold, the value of $\varphi(c)$ for which we can decide if c contained failure sequences or not.

Threshold Determination: Obtaining appropriate detection threshold, τ is important for good result. We treat this as minimization of average percentage error (APE) of *miss-detection* of failure sequences as seen in Equation 4.12.

$$APE = \frac{md + fp}{ns} \tag{4.12}$$

where md is the number of failure detected as non-failure (miss-detection), fp is the number non-failure detected as failure(false positive), ns is the total number of sequences. Our decision space is constrained by setting the threshold values within [0.1 0.9] (*this is because we earlier have normalise our data to values within 0 and 1*). The value with the minimum APE is chosen as our τ. Deferential Evolution (DE) is an approach that is well-studied in other fields [28] and can be used to obtain good detection threshold. However, our decision space is too small to warrant such computationally expensive approach.

4.6 Experiment

The aim of our methodology is to detect patterns that eventually lead to failure. We evaluate the methodology on logs from productions systems. This section explains how we performed the experiments on the logs of two production systems. These are: (1) Syslogs (2) Rationalized logs (Ratlogs), both from Ranger supercomputer logs from Texas Advanced Computing Center (TACC) at the University of Texas, Austin[4] and (3) the logs from IBM Blue Gene/L supercomputer is available on the USENIX repository[5]. A summary of the logs used is shown in Table 3.1. We refer our reader to [24], [60] for more details about

[4] *www.tacc.utexas.edu*
[5] *www.usenix.org/cfdr*

Ranger supercomputer logs and [88] for the Blue Gene/L logs.

4.6.1 Experimental Setup

Failure as well as non-failure patterns from the three system logs are used for testing the methodology. These patterns are obtained after careful consideration of the time between consecutive failures by the experts. In considering the failure data, overlapping failure patterns is avoided. In doing so, the time between two non-overlapping failures events t_{fe_i} and t_{fe_j} must satisfy: $\alpha_t \leq |t_{fe_i} - t_{fe_j}| \leq tw$, where α_t is a small time between two failure events fe_i and fe_j, for which they can be regarded as similar and tw is the time window considered. For example, two nodes may suffer the same correlated failure, only one of such event is considered. Similarly, we obtained non-failure or good sequences.

A maximum time window, tw, of 120 minutes is considered for our experiment, i.e., events that occur within 120 minutes of a failure forms a sequence. We also observe patterns at times (20, 40, 60, 80, 100) in our experiment, with the failure occurring at time .

The choice of 120 minutes time window stems from an established work [25], [59] and the system's administrators advice, they established that the minimum mean time to failure of failure correlated events(Ranger supercomputer) is around 2 hours or more. In order to avoid the probability of overlapping failure runs, we chose the 120 minutes time window. This also is helpful in establishing that detection can be done within a short time period.

The summary of the sequences/patterns is seen in Table 4.1.

Table 4.1: Summary of sequences/patterns obtained from the three production system's logs

Logs/Pattern	***Faulty***	***Non-Faulty***	***Total***
Syslogs	101	207	308
Ratlogs	93	212	305
BGL	42	78	120

4.6.2 Evaluation Metrics

Our goal is to be able to identify sequences or patterns in event logs that are failure-inducing. That is, given patterns, we could separate failure-inducing patterns from non-failure patterns. To measure performance of our detection algorithm, we employ the widely used performance measures in Information Retrieval (IR) namely, *Precision, Recall* and *F-measure metrics.* Precision is the relative number of correctly detected failure sequence/patterns to the total number of detections, Recall is the relative number of correctly detected failure sequences to the total number of failure sequences and F-measure is the harmonic mean of *precision* and *recall* as expressed in Equations 4.13, 4.14 and 5.10 respectively. We capture the parameters in the metrics as follows and as seen in Figure 4.6.

Figure 4.6: Evaluation metrics

- *True positives (TP)*: Number of failure sequences correctly detected.
- *False positives (FP)*: Number of non-failure (good) sequences detected as failure.
- *False negatives (FN)*: Number of failure sequences identified as non-failure sequences.

$$Precision = \frac{TP}{TP + FP} \quad (4.13)$$

$$Recall = \frac{TP}{TP + FN} \quad (4.14)$$

$$F - Measure = \frac{2 * Precision * Recall}{Precision + Recall} \quad (4.15)$$

A good detection approach or mechanism should provide a high value for the metrics above. A *recall* value of 1.0 meaning that the approach can detect every single fault pattern and value of 0 implying the approach is useless as it cannot detect any failure-inducing pattern.

4.6.3 Parameter Setting

The choice of parameter values largely affects the result of any experiment as it is challenging to navigate the large solution space. Likewise, choosing optimal parameter values is practically difficult. We employ experimentation to determine suitable values for our parameters. The thresholds used at the filtering, clustering and detection steps are chosen based on experimenting with several values and choosing the best options. Table 4.3 show a cluster with two sequences/patterns (Seq. 1 and Seq. 2). These patterns are very similar (*sim=0.83*), and are detected as failure sequences. Careful manual observation shows that *seq. 2* is not a failure pattern as it eventually experiences successful recovery with no failure event (hence it is a *FP*), unlike *seq. 1*, which ended in failure (soft lockup).

Naïve Clustering was able to obtain good clusters which represents similar sequences leading to failures as seen from Figure 4.7. This figure shows the validity index (goodness) of the clustering (i.e., how closely related are the sequences). However, with careful consideration of the clusters, we observed that the same sets of events are captured in different clusters due to the frequency

of occurrence of the events. The inter-cluster similarities of these sequences are however not low, averaging around *0.55*. This implies that the tendencies for similar failure sequences to be in different clusters is high, necessitating HAC. This is observed more in both Ratlogs and Syslogs. BGL sequences tend to behave differently with fewer events forming sequence.

The silhouette values obtained for HAC algorithm is slightly lower than the Naïve clustering for the first 100 minutes time windows. In Naïve clustering, only sequences where faults occur at close time interval are clustered together and excludes fault sequences with smaller lead time to failure. On the other hand, with HAC, the temporal effect on sequences is reduced (i.e., patterns are clustered together even when faults occur at different times). This will eventually reduce the value of validity index. This scenario is not seen in BGL because different temporal occurrence of faults that changes patterns is rarely observed.

Table 4.2: Experiment Parameter Values

Threshold Parameter	λ	δ	τ
Parameter Value	2	0.6	0.2

Hence, based on all these scenarios explained above, we adopt experimentation on several possible parameters to enable us choose the best parameters as seen in Table 4.2. For our filtering approach, few parameters were used; the event's similarity threshold $\lambda = 2$. This value was chosen as explained in Section 4.3.2. The sequence clustering *similarity threshold* value, δ for both Naïve and HAC clustering is **0.6**. At this similarity value, we obtained a better *validity index* (goodness of cluster) for the clustering, see Figure 4.7. Detection threshold, τ, the value for which we can decide if a sequence is a failure or not was set to be $\tau = 0.2$. This value is obtained by performing detection using values within the range [0.1 0.9] while we observe the *average percentage error (APE)* of miss-detection given by equation 4.12, Figure 4.8 shows that the lowest APE is obtained at the detection threshold $\tau = 0.2$, hence, this informed

our choice.

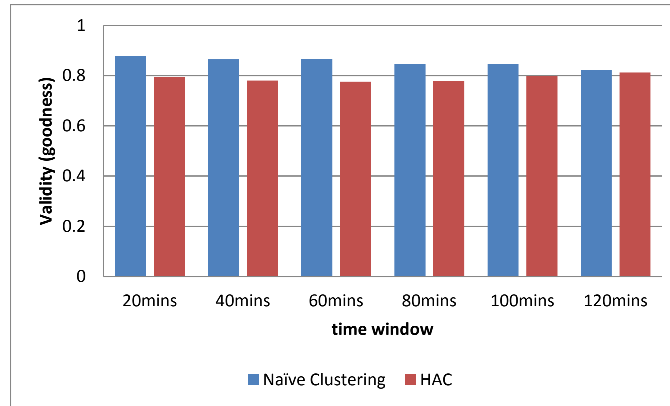


Figure 4.7: Cluster goodness based on intra-cluster and inter-cluster similarity (on Syslog, JSD metric)

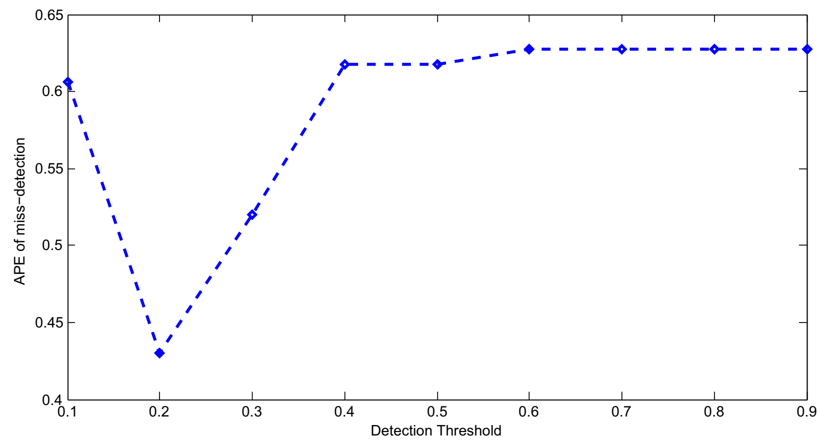


Figure 4.8: APE (percentage miss-detection) vs Detection Threshold

Table 4.3: Sample Clustering Result (syslog, HAC) for a cluster with sequences (seq.1 and seq.2)

	Sim	**Sequence Events**
Seq. 1	0.83	Machine check events logged error occurred while communicating with,**@o2ib ** Connection service share OSTd via nid,** @o2ib ** was lost progress operations using this service will wait for recovery complete Skipped previous similar messages error occurred while communicating with,**@o2ib ** OSTd_UUID not available for connect stopping Request sent from share OSTb NID,** @o2ib ** ago has timed out limit, quota_interface still havent managed acquire quota space from the quota master after retries err rc Machine check events logged, Connection service share OSTd via nid **@o2ib** was lost progress operations using this service will wait for recovery complete Recovery timed out ** **BUG soft lockup detected CPU** pid uid comm ldlm_bl_**
Seq. 2		Machine check events logged Skipped previous similar messages error occurred while communicating with,**@o2ib ** Connection service share OSTd via nid,**@o2ib ** was lost progress operations using this service will wait for recovery complete Machine check events logged error occurred while communicating with,**@o2ib ** Connection restored service share OST using** **Recovery complete with ****, Machine check events logged

4.7 Results

We show the results of the experiments conducted and we also discuss these results in this section.

Filtering

We show from Figure 4.9, the rate of compression of logs under varying values of LD threshold, λ. At $\lambda = 2$, The filtering approach obtained good compression from original logs. It achieved compression of 78%, 80% and 84% on syslog, ratlog and BGL logs respectively.

Figure 4.9: Compression rates given varying LD on syslog

Normal filtering achieved an average compression of 88%, which is higher than our method. Both *normal filtering* and Zheng *et al.* [154] approaches obtain better compression rate because their methods was able to filter events deemed similar within given time. Unlike theirs method, we believe that doing so will remove important events that could serve as precursor or signatures to failure. Hence we kept such causally-related but semantically unrelated events.

Figures 4.10, 4.11 and 4.12 demonstrates the result of performing detection (*using HAC, correlation distance*) on logs filtered using the *normal filtering* and *our filtering method*. The results show that our method achieved an average *F*-

measure of 77% while normal filtering achieved 48% on *syslog*. An improvement of about 19% over the *normal filtering* is achieved on ratlog and 10% on BGL. The implications of these results is that filtering based on our approach greatly enhance failure patterns clarity by preserving useful failure precursor events.

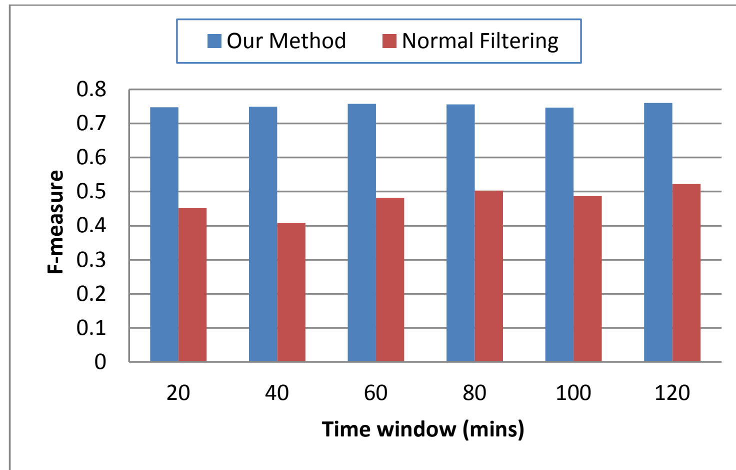


Figure 4.10: Showing the *F-measure* detection of both *our method* and *normal filtering* on syslog

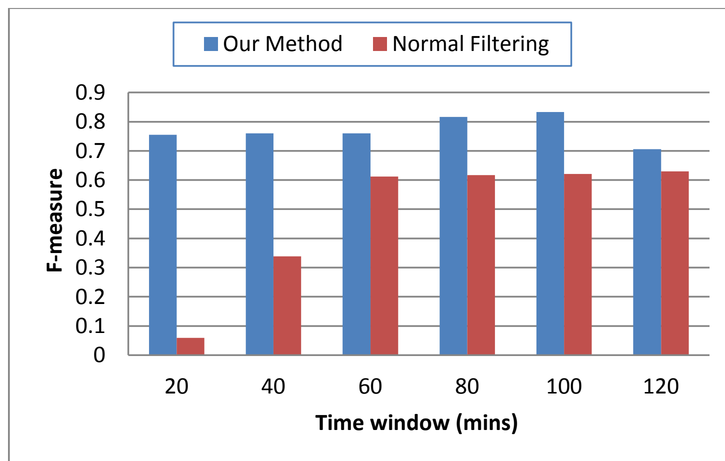


Figure 4.11: Showing the *F-measure* detection of both *our method* and *normal filtering* on ratlog

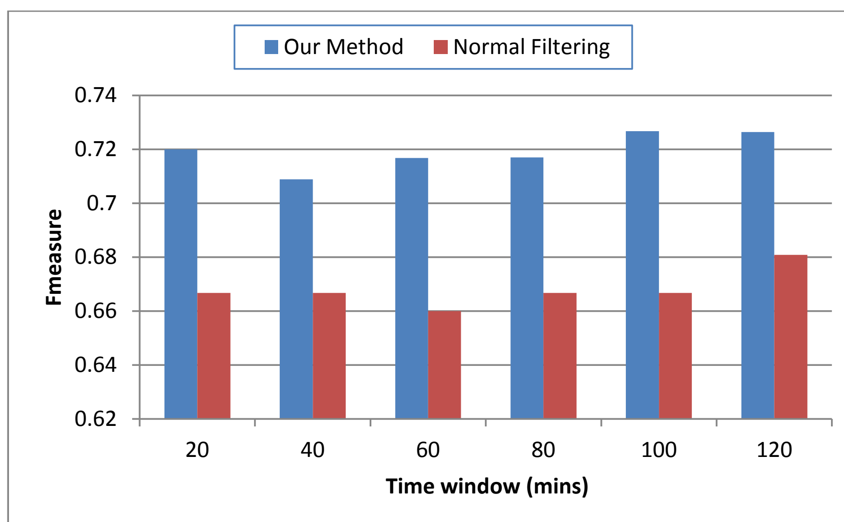


Figure 4.12: Showing the *F-measure* detection of both *our method* and *normal filtering* on BGL

Filtered and Redundant Logs

Preprocessing, especially filtering redundant events in logs is not a compulsory step in log analysis. However, it is necessary when there is need. It should not reduce the effectiveness of the targeted analysis. To demonstrate the effectiveness of our redundant event filtering for failure pattern detection, we conducted experiments on both logs without redundant event removal and the filtered logs. The result is as shown in Figures 4.13, 4.14, 4.15. It also contains using only the ids as representative of events for detection.

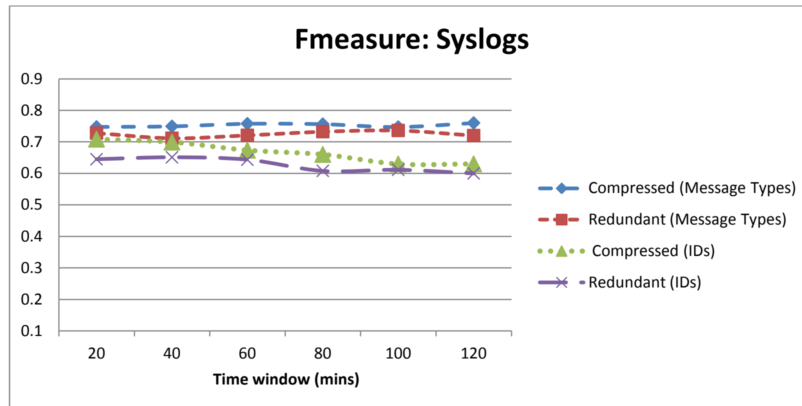


Figure 4.13: Showing the *F-measure* detection on both *filtered* and *redundant* logs (syslog)

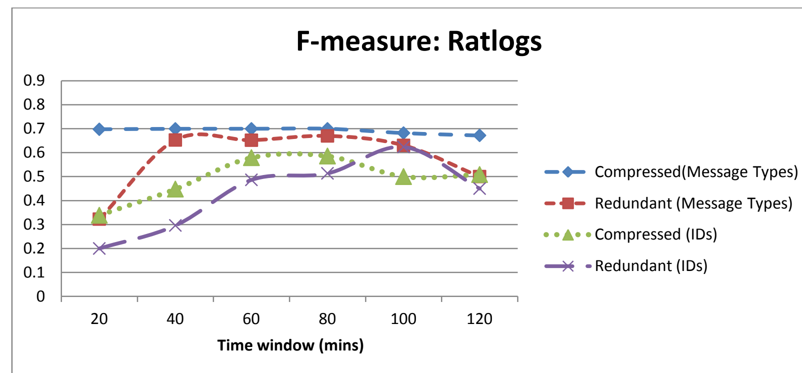


Figure 4.14: Showing the *F-measure* detection on both *filtered* and *redundant* logs (ratlog)

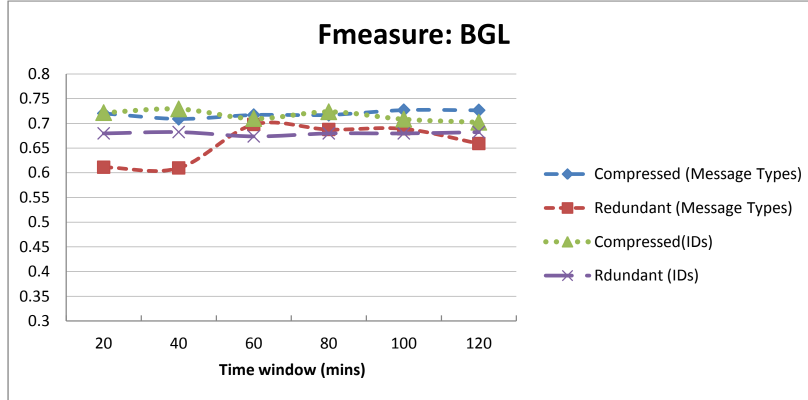


Figure 4.15: Showing the *F-measure* detection on both *filtered* and *redundant* logs (BGL)

Existing Detection approach

We conducted an experiment to evaluate how our method perform compared to Xu et al. [147] approach. We choose this for comparison because it is a well-established method in the field and related to our method.

In summary, **Xu**'s approach basically creates features from the console logs. Feature vector is created base on the frequency of events, analogous to the concept of Bag of Words (BoW) in information retrieval. Detection of fault events is treated as anomaly detection of the feature vectors which are labelled as faulty or non-faulty. Their approach further uses principal component analysis (PCA) to perform anomaly detection or subsequent detection of faults. Compared to our approach, we created feature vectors which form inputs to our algorithm the same way. However, the features we used are event types in sequence. we performed detection differently, we do not employ PCA, and we developed an algorithm that utilises the entropy and mutual information of the sequences to detect failure features. Each sequence which is a features is labelled as faulty or non-faulty. We do not detect individual messages, rather, we detect group of messages that form sequence which are symptomatic to failure. This necessitated the idea of clustering in our algorithm. We explain the results for both

methods in next section.

Pattern Detection Results

The results we present here compares our approach to detecting failure patterns in logs with an existing approach we called **Xu**.

The legends of the graphs in Figures 4.16, 4.17, 4.19,4.20, 4.22 and 4.23 indicates: clustering algorithm and distance metric used. For example, *HAC.Corr* implies *HAC algorithm* with *Correlation distance*.

Syslog: The performance of the approach when applied on syslog was good. A high detection is achieved across all the time windows as seen in the values of *recall* and *precision* in Figures 4.16 and 4.17 respectively. Particularly using *HAC.Corr* achieved a recall of more than 90% across all time windows. This implies that on syslog data, irrespective of the time window used for detection, our approach can achieve high detection. Meanwhile, Xu's method achieved low performance seen in the F-measure values (Figure 4.18); it only improve at higher time window (120 minutes). However, our approach performed consistently better across all the time windows used. Looking at the two clustering approaches used, the performance using both HAC and Naive clustering are consistently high, (see Figures 4.16 and 4.17), with average precision of 65% and recall of 88%. Using correlation distance metric (Corr), it performed slightly better than using JSD as time tends towards failure (see Figure 4.18). This implies that detection of failure sequence is most effective using our method when correlation between interacting components of system is high.

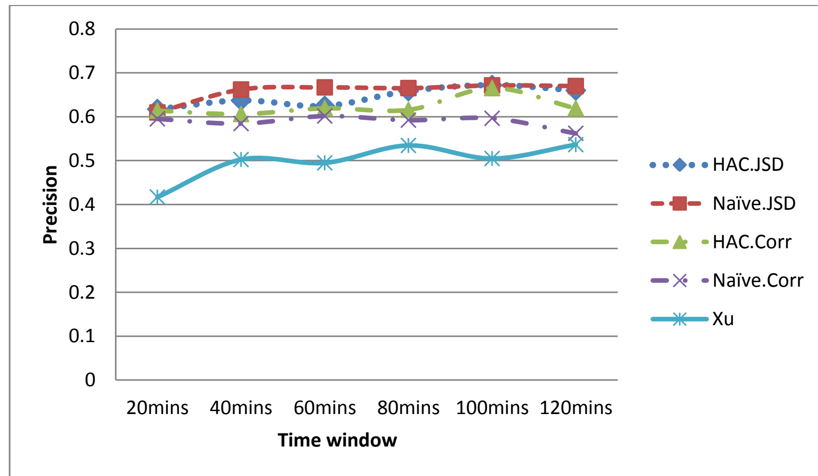


Figure 4.16: The *Precision* of our failure pattern detection and *Xu's* method on *syslog*

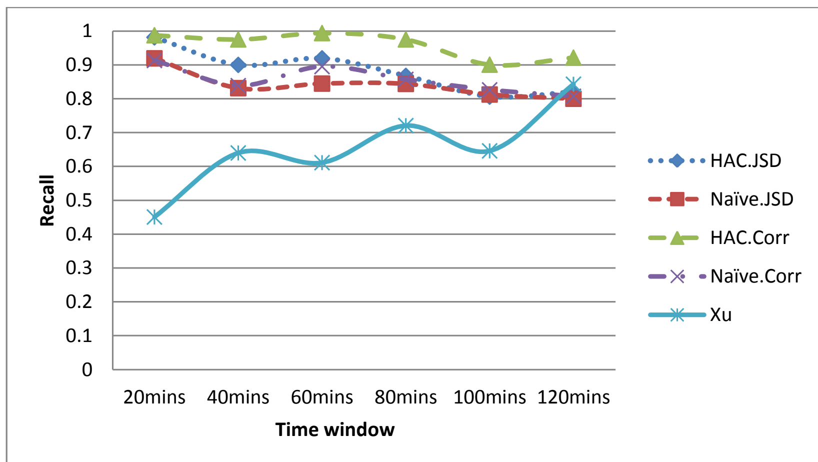


Figure 4.17: The *Recall* of our failure pattern detection and *Xu's* method on *syslog*

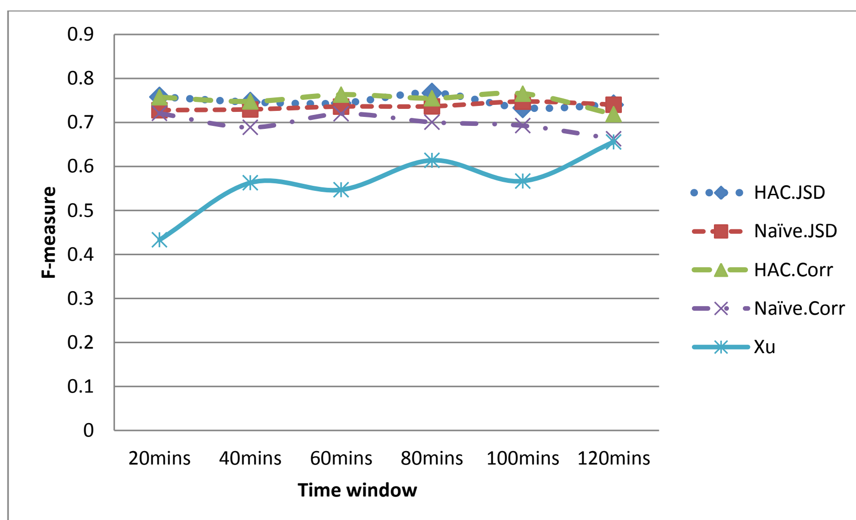


Figure 4.18: The *F-measure* of our failure pattern detection and *Xu's* method on *syslog* data

Ratlog: Performance of our method on **ratlog** is very good compared to Xu's method as can been seen in the *precision* and the *recall* values of Figures 4.19 and 4.20. We obtained up to 79% precision with the use of correlation distance and recall above 90% for *HAC.Corr*. Naive clustering approach performed poorly at lower time windows. Effectively, from Figure 4.21, the result demonstrates that F-measure is high for our method for all the time windows. On the other hand, Xu's approach achieved its best F-measure at higher time window of 120 minutes. Our method achieved better F-measure because the logs contains better event information characterizing faults and the lead time to failure is small, implying many different faults patterns could be experienced late towards failure.

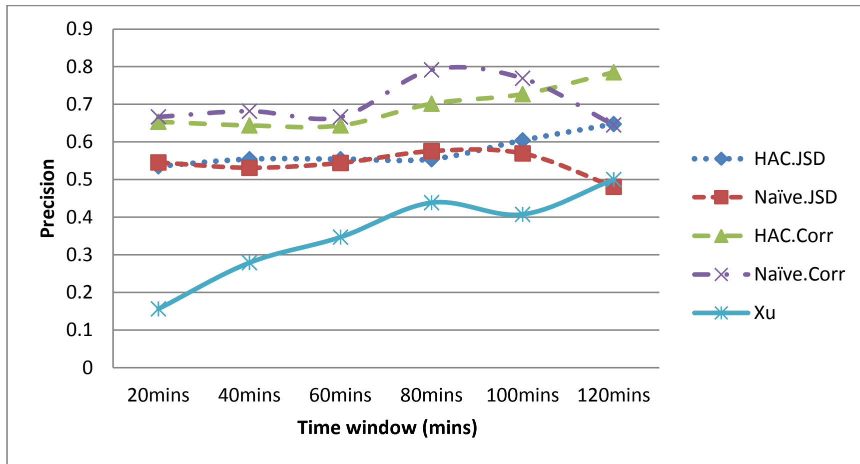


Figure 4.19: The *Precision* of our failure pattern detection and *Xu's* method on *ratlog*

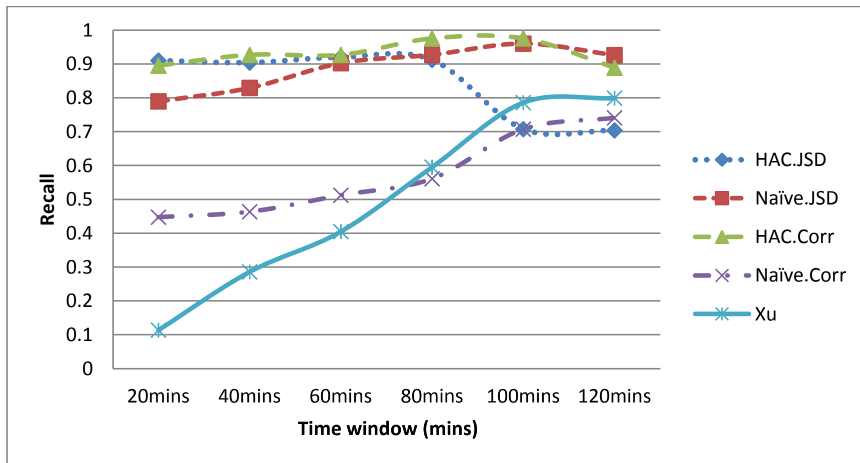


Figure 4.20: The *Recall* of our failure pattern detection and *Xu's* method on *ratlog*

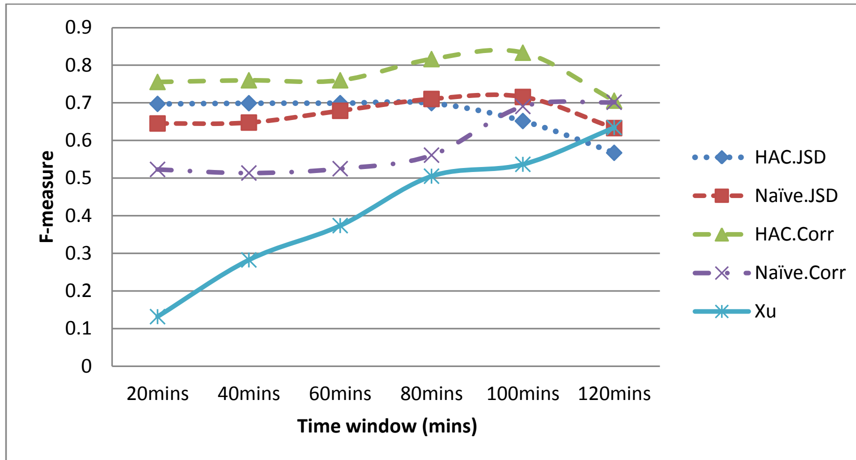


Figure 4.21: The *F-measure* of our failure pattern detection and *Xu's* method on *ratlog*

BGL Results on **BGL** logs seen in Figures 4.22 and 4.23, compared to other two logs, is relatively inconsistent in its detection performance across all the time widows used. However, the performance of our method is better than Xu's approach. It achieved an average *recall* and *precision* improvement over Xu's method by about 15% and 22% respectively. This is equally reflected in the F-measure values seen in Figure 4.24. Based on our investigation, we observed that BGL logs rarely reports fault within short time window and where there are some failure events, there could be no or few preceding precursor events that can be observed. This could be the reason for relatively inconsistent detection performance. A more longer time window could show better result.

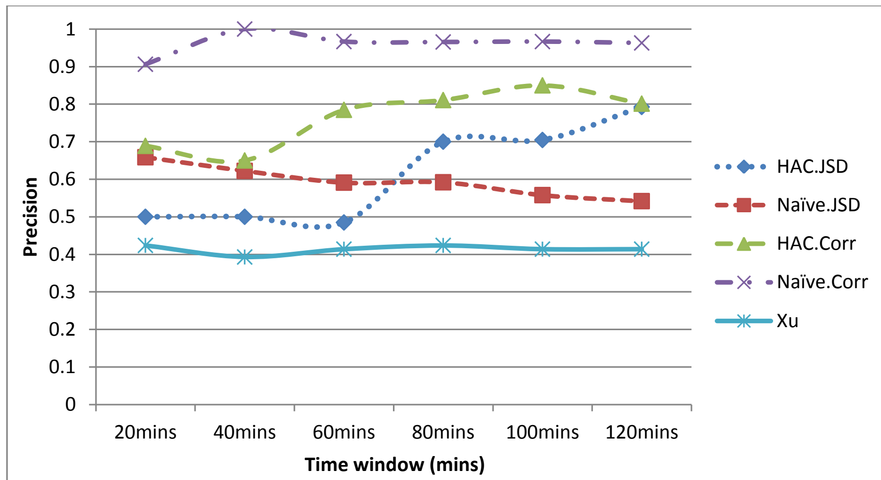


Figure 4.22: The *Precision* of our failure pattern detection and *Xu's* method on *BGL*

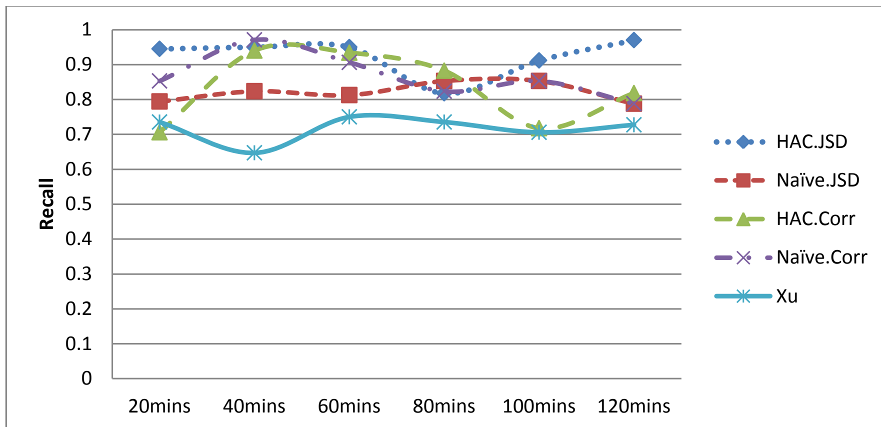


Figure 4.23: The *Recall* of our failure pattern detection and *Xu's* method on *BGL*

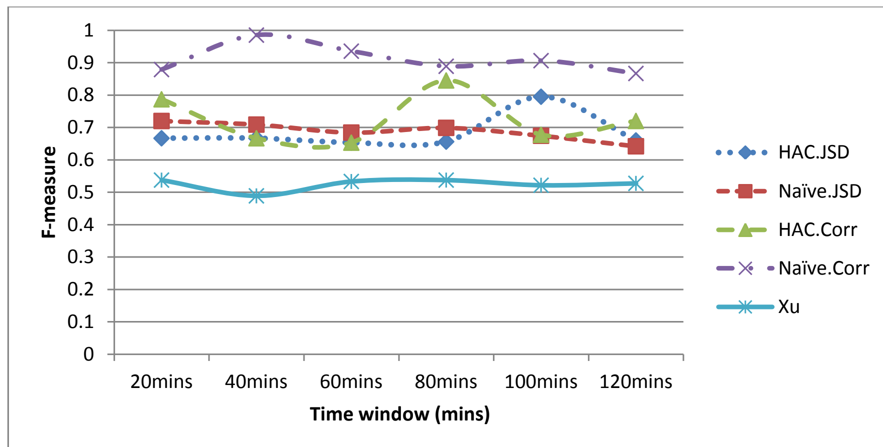


Figure 4.24: The *F-measure* of our failure pattern detection and *Xu's* method on *BGL*

4.7.1 Runtime Analysis

The execution time of *detection algorithm* is small across all the time windows (highest about 33 seconds) as seen in Figure 4.25. It scales increases with increase in log size and time window. This is good, and detection is achieved in good time. Obviously, the execution time is dependent on the size of logs and the larger the time window, the bigger the size of logs.

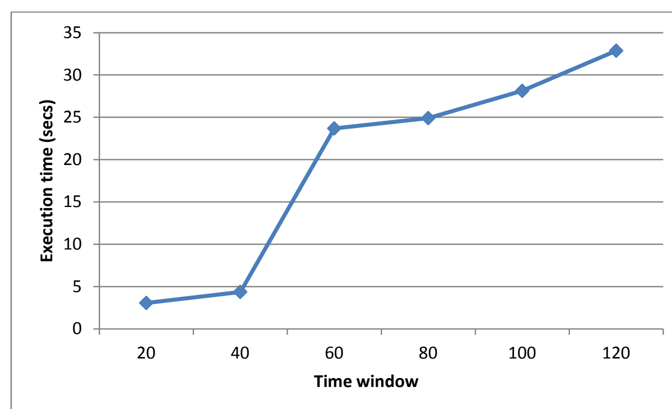


Figure 4.25: The runtime graph of Detection approach

4.8 Summary

The contributions of this chapter are as follows: Firstly, we enable failure prediction on the basis of observed patterns collected in an unsupervised manner. Secondly, the high computational cost associated with recovery is only incurred when there is an impending failure. We first filter event logs based on their *similarities*, in order to preserve patterns deemed useful or are most indicative of as a precursor to failures. Subsequently, using the filtered logs, we develop an unsupervised approach that leverages fault characteristics to detect patterns leading to a failure among runs of event logs.

Technically, We proffer a log-analysis methodology that first filters the log files and subsequently propose an error detection methods based on pattern similarity. We apply our approach on log files obtained from the Ranger supercomputer and the IBM BlueGene/L and obtained a F-measure on detection.

CHAPTER 5

Improving Error Detection Using Resource Usage Data and Event Logs

5.1 Introduction

A number of works [9, 43, 53, 112, 147] have shown that detecting fault occurrences (i.e., errors) using logs and other data has received good attention, with reasonably good results as seen in the literature review. The challenge is that, even though the earlier mentioned attempts reported good detection results, they are constrained mainly to failures which are remarkably characterised by frequent reporting of events, as seen in Figure 5.1. However, this is not the case for some types of failures experienced in HPC systems. For example, some failures caused by soft errors may not produce visible and abnormally frequent events that would signify a faulty behaviour. More so, some faults can induce a silent behaviour (i.e., events are sparse) and only become evident when the failure is about to occur (see Figure 5.2). In such a case, this provides a small mean time to failure in the sense that the time between the error detection and the actual failure is small. In these cases, approaches based on entropy and mutual information, as mentioned above, will not help.

In general, the event logs provides insufficient information regarding the behaviour of HPC systems that can be used to accurately detect errors in the system. To circumvent the problem induced by the incompleteness issue of event logs, we complement these logs with *resource usage* data in this study. In short, resource usage data captures the amount of resources consumed or produced by all the jobs in the system. For example, the resource usage data file may contain the amount of memory a job is using at a given time as well as the number of

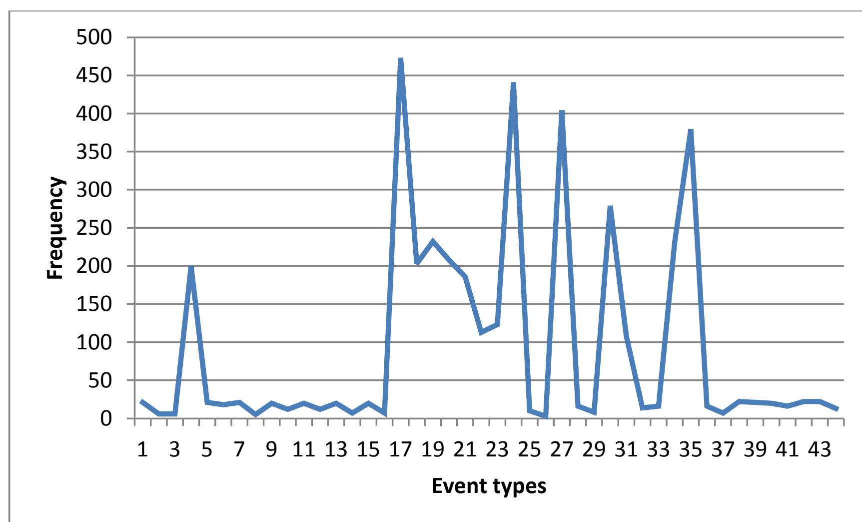


Figure 5.1: *Bursty faulty event sequence* showing the behaviour of fault events logged within an hour

messages a job has output on the network. It has been generally argued that unusually high or anomalous resource usage could lead to failure of systems [25]. Most supercomputer systems do not make such usage data available, making it difficult for researchers to verify this.

To detect errors in the system, we propose a *novel* approach that uses both event logs and resource usage logs. The use of resource usage data enables the detection of anomalous jobs which, when coupled with the erroneous behaviour of nodes (as captured from the event logs), will indicate the existence of an error in the system. The resource utilization would provide us with better understanding of the system's behaviour in the face of few precursor events in the event logs. The approach is *unsupervised*, meaning that neither the resource usage data nor event logs carry labels. This is valuable and relevant for two reasons: (i) when resource usage data is used, given that little work has been done based on resource usage data, very little labelled data is available and (ii) the ability to provide labelled data requires extensive and detailed knowledge about the system, which may not always be available; additionally, the labelling

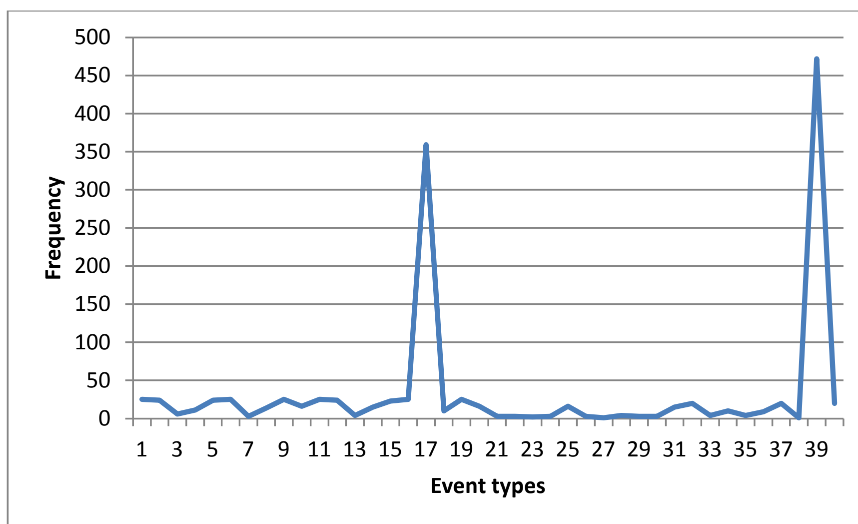


Figure 5.2: *Silent faulty event sequence* showing the behaviour of fault events logged within an hour

process is also time-consuming.

The objective of this chapter can be stated as follows: Given an event sequence and resource usage data, we seek to determine whether the event sequence contains error messages that indicate impending failure. We seek to find an unsupervised approach to detecting patterns indicative of failure from both event logs and the resource usage data of any HPC system.

5.2 Detection Methodology

We now briefly summarize our methodology. Firstly, event logs are reported as stream of events occurring in time. However, to keep the log analysis tractable, it is beneficial to break a long sequence into smaller sequences of similar size, which we measure in time units (obtained from the timestamps). We thus call the size of a small sequence the *time window*. The choice of the time window is dependent on how informative events within such time window are, i.e., the time window needs to be as small as possible but with enough informative events to

characterise the time window. Given that event logs contain both normal and error messages [43] and are generally incomplete, we seek to complement these event logs with resource usage data to aid error detection. Here, our premise is that abnormal resource usage is indicative of a fault having occurred in the system,

Next we transform these data into a format that captures the system behaviour within the chosen time window and can be used easily by any detection algorithm. We create a feature matrix from the event log and another from the usage data; we then extract the *Mutual Information* and *Entropy* of the interacting nodes and event types respectively. This is done after nodes with similar behaviour are clustered together. We then determine "outlierness" of a sequence by performing a PCA outlier detection on the resource utilisation data feature matrix. This step provides us with an anomaly score of a sequence which is then used in error detection. The detection process leverages the mutual information, entropy and anomaly score of the sequence to determine whether the sequence is likely to end in failure. This section focuses on detailing our methodology as seen in Figure 5.3.

5.2.1 Data Transformation

Notations and Terminologies: Before explaining our methodology, we briefly mention the notations used. We will denote a specific item using a small letter, while we will use the associated capital letter to denote the total number of that item. For example, we will denote a specific node by n (or n_i) and the total number of nodes in the system by N.

This section focuses on obtaining appropriate system data that can represent nodes and running jobs behaviour. These behaviour inherently describes the state of the system within a given time. The data (both event logs and resource usage data) need to be transformed into a format suitable for analysis. In order to detect abnormalities in resource usage, we extract the attributes/elements which capture the state of the resources for each job on each node within given

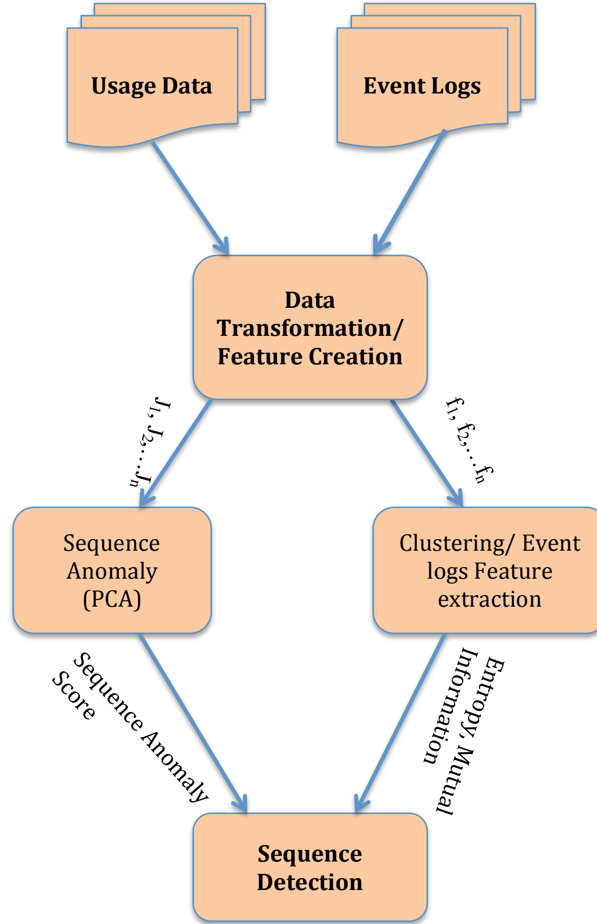


Figure 5.3: Methodology work flow showing steps taken to achieve detection

time. Generally, we extract features from event logs and then from resource usage data.

Event logs Features

For event logs, we denote by e_i^l the number of events e_l produced by node n_i within a given time window. Given E different possible events, we obtain a vector $[e_i^1 \dots e_i^E]$ which contains the number of occurrences of each event on node n_i. We call this vector an *event feature* of node n_i. This concept is analogous to the bag of words concept used in information retrieval, which has been proven

to be effective in capturing relationships between terms/words/messages. We reuse the same idea because the event log messages are our primary source of data about the health of the systems. The features that are produced depend on the frequency distribution of event logs. Hence, for a given time window t_w, each node will produce an event feature, resulting in a matrix, where each row represents a event feature and each column represents the number of occurrences of a given event by different nodes in the system.

For a given time window tw, we denote the resulting *event feature* matrix by F_{tw}, as shown in Figure 5.4.

$$F_{t_w} = \begin{pmatrix} e_1^1 & e_1^2 & . & . & . & e_1^E \\ e_2^1 & . & . & . & . & . \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ e_N^1 & e_N^2 & . & . & . & e_N^E \end{pmatrix}$$

Figure 5.4: Data matrix F_{t_w} of a sequence with N nodes and E event types, where e_i^l denotes the number of occurrences of event e_l by node n_i.

Resource Usage Features

A system will typically have a number of counters that capture different aspects of the system during execution. These counters indicate the amount of resources they are associated with that are being produced or consumed. For example, a memory counter may capture the amount of memory that is being used by a given job at a given time. Denoting a given job by j_k which executes on node n_i during time window tw, we denote the amount of a resource r used by j_k on n_i during tw by $U_{k,i}^{r,w}$. We call the vector of resources used or produced by a given job j_k on node n_i during time window tw as a *resource feature* and we denote it by $U_{k,i}^{w}$ and is defined as $U_{k,i}^{w} = [U_{k,i}^{1,w}, \ldots U_{k,i}^{R,w}]$. Whenever the exact location of a job is unimportant, we will denote the usage matrix by U_k^w, i.e., U_k^w denotes the different resources used or produced by job j_k in time window tw.

Similar to event features, we construct a $J \times R$ *resource feature* matrix[1] for a given time window tw, which we denote by U^{tw}, where each row is a resource feature of a job and each column is the vector that captures the amount of the specific resource used or produced by all the jobs in the time window. Specifically, the value associated with $U^{tw}[k, r]$ represents the amount of resource r that has been used or produced by job j_k during time window tw.

5.2.2 Event Clustering and Feature Extraction

Given the event features created from the *event logs* as depicted by the matrix of Figure 5.4, we perform clustering to group nodes exhibiting similar behaviour. The underlying reason for this is that, nodes executing similar jobs tend to log similar events. This means that a component fault could get to be reported by these nodes enabling us to capture node level behaviour of the systems through clustering. Mutual Information and Entropy of the individual features clustered are obtained.

Clustering

Makanju et.al., [94] has shown that the state of systems can be discovered through information content clustering. As stated earlier, Oliner et al., [106] verified the hypothesis that similar nodes correctly executing similar jobs should produce similar logs. This implies that a fault in a node or job could cause a cascading effect or alter communicating jobs, which could result in producing similar fault events. This also means that we can leverage the homogeneity of large scale systems to improve fault detection. To achieve this, we employ clustering to group nodes with similar behaviour to be able to extract the group behaviour.

Clustering groups similar data points together in such a way that those in different group are very dissimilar. In this case, we group nodes with similar behaviour in terms of the events they log within the given time window. Hence,

[1] U can be used to denote usage matrix when the tw is not used for purpose of clarity

given a sequence of events within a given time window, the event features which are formed from the data transformation section is an $N \times E$ matrix, F_{tw}, (see Figure 5.4) with N being the number of nodes (rows) and E the number of event types (columns). e_i^l is the count of event types e^l produced by node n_i. We employ a simple centroid-based hierarchical clustering (see Algorithm 4) to perform this. We employ this clustering approach because we assume there are one or more outlier nodes within a sequence, and a centroid-based clustering as this is not sensitive to outliers.

In any distance-based clustering, the distance metric is key to achieving a good result. For our purpose, we defined a distance metric that could capture the informativeness of each node within the sequence and/or the correlation between the events types in the sequence. Hence we employ the *Correlation Metric* as our clustering similarity distance metric.

Correlation Metric (Corr), is based on the correlation within nodes of the sequence. Given two feature vectors of F_{tw} as x_i, x_j, then we compute the similarity as follows:

$$\mathrm{cov}(x_i, x_j) = \frac{1}{N} \sum_{k=1}^{N} (x_i^k - \overline{x_i})(x_j^k - \overline{x_j}) \tag{5.1}$$

$$std(x_i) = \sqrt{\frac{1}{N} \sum_{k=1}^{N} (x_i^k - \overline{x_i})^2} \tag{5.2}$$

where $\overline{x_i} = \frac{1}{N} \sum_{k=1}^{N} x_i^k$ is the mean of vector x_i; then,

$$sim(x_i, x_j) = 1 - \left| \frac{\mathrm{cov}(x_i, x_j)}{std(x_i) * std(x_j)} \right|. \tag{5.3}$$

We treat $sim(.)$ as a similarity or distance measure between two feature vectors.

The above distance metric must and is observed to satisfy the following properties given two variables x_i and x_j:

1. Non-negativity property, that is, $0 \leq 1 - |sim(x_i, x_j)| \leq 1$.

2. $1 - |sim(x_i, x_j)| = 0$ if and only if x_i and x_j are linearly related or x_i is same as x_j.

3. Symmetric: that is $1 - |sim(x_i, x_j)| = 1 - |sim(x_j, x_i)|$.

4. Scalability: for some given constants c_1, c_2, c_3, c_4, if $y_i = \frac{x_i - c_1}{c_3}$ and $y_j = \frac{x_j - c_2}{c_4}$, then $1 - |sim(x_i, x_j)| = 1 - |sim(y_i, y_j)|$. In essence, the measure is invariant to scaling and variable translation.

Algorithm 4 An algorithm for clustering similar behaving nodes

```
1:  procedure CLUSTERING(ℂ, λ)
2:      Inputs: |ℂ| event features with c_i ∈ ℂ, i = 1...n; MinimumSimilarityThreshold λ.
3:      Initially assume each node vector as unique, with each as a cluster on its own;
4:      for i = 1 to |ℂ| − 1 do
5:          for j = i + 1 to |ℂ| do
6:              sim = sim(c_i, c_j)     ▷ Find similarities of centroids of c_i and c_j
7:              if (sim ≥ λ) then
8:                  Merge c_i and c_j;
9:              end if
10:         end for
11:         add the merged cluster to set of new clusters
12:     end for
13:     Repeat step 1 with new set of merged clusters until similarity less than λ is achieved
14:     Return new m sets of clusters of similar nodes
15: end procedure
```

Sequence Feature Extraction

The amount of information provided by the nodes within a given time window and also the informativeness of the event types could provide useful hints on unusual behaviour of such nodes within the time window. We assume that these behaviours can be succinctly represented by these two features of the sequence: *Mutual Information* (I) and *Entropy* (H) of event types produced by each node within the given time window. To further support our approach, it has been argued that changes in entropy are good indicators of changes in the behaviour of distributed systems and networks [79]. This informed our decision

to extract these event features. We assume that a higher uncertainty (entropy) with reduced mutual information could signify abnormal system behaviour or a failure sequence with the converse being true for normal behaviour.

Sequence Mutual Information (I)

Mutual Information measures the relationship between two random variables especially when they are sampled together. Specifically, it measures how much information a random variable e_i (event feature) carries about another variable e_j. Essentially, it answers the question about the amount of information event features e_i and e_j carry about each other. Hence, given cluster of event features $\mathbb{C} = \{c_1, ..., c_m\}$ and $c_i = \{e_1, ..., e_n\}$, containing set of similar event features, then,

$$I(c_i) = \sum_{j=1}^{N-1} \sum_{k=j+1}^{N} p(e_j, e_k) \log \frac{p(e_j, e_k)}{p(e_j)p(e_k)} \tag{5.4}$$

where $p(e_j, e_k)$ is the joint distribution of event features of cluster c_i, $p(e)$ is the marginal distribution of event feature e. Then. $I(\mathbb{C})$ is given by: $I(\mathbb{C}) = \frac{1}{|\mathbb{C}|} \sum_{c_i \in \mathbb{C}} I(c_i)$. Nodes within a sequence characterised by frequent events logging and probably similar events due to the same fault will have high mutual information.

Sequence Entropy (H)

Entropy, on the other hand, is a measure of uncertainty of a random variable. In other words, when the information content of event types of a sequence are highly unpredictable, then the sequence has high entropy. For each event cluster c_i,we define entropy as follows:

$$H(c_i) = -\sum_{j} p(e_j) \log p(e_j) \tag{5.5}$$

where $p(e_j)$ is the distribution of event types of event features in c_i. The entropy of the sequence is then given by the average of the entropies of each cluster of the sequence. The changes observed in entropy are usually good indicators of

changes in the behaviour of the cluster system. Such changes may point to imminent failure in the cluster system.

5.2.3 Jobs Anomaly Extraction from Resource Usage Data

In a cluster system, large amount of nodes are present with hundreds of jobs running. The resource usage data contains statistics about the usage of resources (e.g. CPU, I/O transfer rates, virtual memory utilization) of each of these nodes by each job running in the cluster system within the given time. This information provides useful hints regarding an unusual behaviour of a given job in terms of its rate of resource utilization. Hence, the resource data transformed into a matrix as explained in section 5.2.1 is used as our input data. Our aim here is to obtain anomalous jobs which could significantly point to problem(s) in the cluster system as observed by the anomalous jobs within a period. A column vector of the matrix represents a job and a row represent a counter or attributes (e.g. *dirty_pages_hits, read_bytes, tx_msgs* etc), see Table 3.2 for the full list of counters.

There are a significant number of research work on approaches to detecting anomalies (see Chapter 2). Since we do not have data labelled with how normal behaviour of jobs should be, the wisest option is to use an unsupervised approach to detect anomalous jobs. Principal component analysis (PCA), has been a widely and efficiently used feature extraction method in different fields [84] and also for identifying anomalous node behaviour [80]. We utilise this approach to find anomalous jobs in resource usage data.

Principal Component Analysis (PCA)

In this section, we briefly explain PCA which would introduce us to how it is utilized for anomaly detection purpose. PCA is a well known and researched technique for dimensionality reduction. PCA basically determines the principal direction of a given data points by constructing the covariance matrix of the data and finding the dominant eigenvectors (principal directions). These eigenvectors

are also seen as the most informative of the original data space. Let $U = [j_1, j_2, ..., j_n]^T \in \Re^{n \times k}$, where each row j_i representing $k - dimension$ data instance of job, and n is the number of data instances or jobs, then PCA, formulated as an optimization problem, first calculates the covariance matrix C of U as:

$$C = \frac{1}{n} U U^T. \quad (5.6)$$

Then the *eigenvalues*, λ_i, ($i = 1...p$ and $p < k$) are calculated and sorted in decreasing order with the first being the highest eigenvalue. From this, a projection matrix $V = [a_1, a_2, ..., a_p]$ consisting of $p-$dominant eigenvectors is constructed, with each data point, x, projected into a new subspace as:

$$X = V^T J. \quad (5.7)$$

The p-dominant eigenvectors produced are in decreasing order of dominance.

Anomalous Jobs Detection in Usage Logs Using PCA

This section seeks to look at utilizing PCA to identify anomalous jobs running in cluster system within a given time window. We employ an approach with the similar property of principal directions as the one made in [84]. The basis for the approach is that, for every data point, removing or adding it contributes to the behaviour of the most dominant direction. This is because PCA relies on calculating mean and data covariance matrix in obtaining eigenvectors, and it is observed to be sensitive to the presence of an outlier. Hence, in this approach, the *outlierness* of a job can be determined by the variation in the dominant principal direction. Specifically, by adding an outlier the direction of dominant principal component changes, but a normal data point doesn't change it.

Sequence Anomaly

The assumption here is that, any anomalous job will cause deviation from the leading principal direction, therefore, a sequence anomaly or "outlierness" is the average value of all the outlier or anomalous jobs present within such time

window. For example, in Figure 5.5, the points in red are likely anomalous jobs.

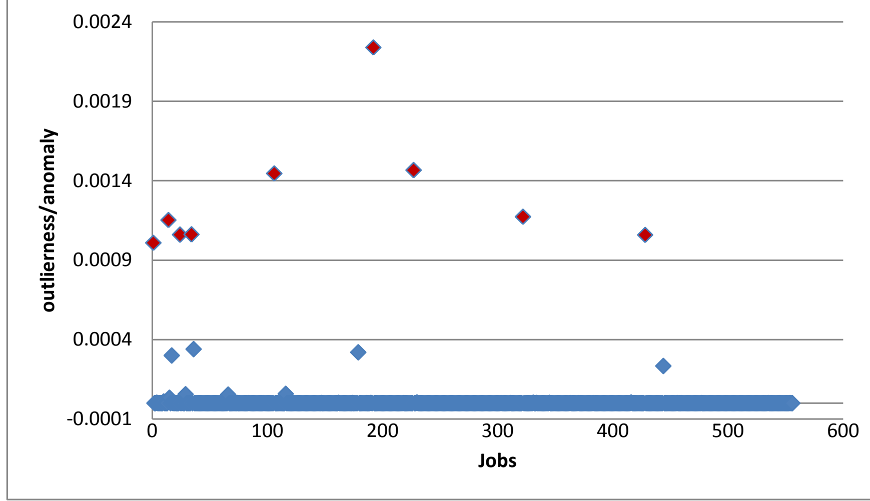


Figure 5.5: Jobs outlierness of a sequence using PCA

So, given data points $U = [j_1, j_2, ..., j_n]^T$, the leading principal direction, d, of matrix U is extracted. Then for each data point j_i, obtain the leading principal component, d_i, of U without data point j_i. The outlierness, a_i, of j_i is the dissimilarity of d and d_i. The algorithm can be seen in Algorithm 5.

Algorithm 5 Sequence anomaly score algorithm

1: **procedure** AnomalyScore(U, γ, j_i)
2: d = Leading Principal direction of J
3: d_i = Leading Principal direction of J without data point j_i.
4: $a = 0$ ▷ initialise anomaly score
5: $sim(d_i, d) = \dfrac{d_i \cdot d}{\|d_i\|\|d\|}$
6: $a = 1 - |sim(d_i, d)|$
7: **if** $(a \geq \gamma)$ **then**
8: return a_i ▷ returns anomaly score
9: **else**
10: return 0 ▷ returns 0 for not anomalous
11: **end if**
12: **end procedure**

Any data point with "outlierness" greater than given threshold γ is regarded as *anomalous*.

5.2.4 Detection of Failure Patterns

The algorithm is aimed at detecting a sequence of events within a chosen time window that most likely indicates the presence of errors. The algorithm leverage the features that capture the behaviour of nodes within a sequence (i.e., mutual information and entropy), where high values (i.e., above a threshold) of these features indicate the presence of errors which would likely lead to a failure. Together with a high $I(s_i)$ and $H(s_i)$, an anomalous sequence a_i of resource usage is a high indication of failure. Otherwise, we consider such a sequence to be normal. The detection algorithm will then depend on our definition of anomaly, as captured by the thresholds. We define thresholds for *Mutual Information* (μ), *Entropy* (φ) and sequence *anomaly*,(γ). The algorithm can be seen in Algorithm 6.

Algorithm 6 Sequence detection

```
 1: procedure DETECT(S, γ, μ, φ)
 2:     for each sequence s_i in S do
 3:         I(s_i) =Mutual Information(s_i)
 4:         H(s_i) = Entropy(s_i)
 5:         a_i = AnomalyScore(S, γ, s_i) ▷ from resource Usage data of sequence
 6:         f_i = I(s_i) − H(s_i)
 7:         if (I(s_i) >= μ && H(s_i) >= φ && a_i > 0) || (a_i > 0 && H(s_i) >=
    φ && f_i <= 0) then
 8:             Return True                                  ▷ faulty Sequence
 9:         else
10:             Return False                   ▷ non-faulty or normal sequence
11:         end if
12:     end for
13: end procedure
```

In summary, the Algorithm 6 detects errors in a sequence based on I, H and sequence anomaly score, a_i.

5.2.5 Experiment and Results

We evaluate our approach through experiments conducted on Rationalized logs (ratlogs) and resource usage data from the Ranger Supercomputer from the

Texas Advanced Computing Center (TACC) at the University of Texas at Austin[2].

The resource usage data were collected using TACC_Stats [59] that takes snapshots at the beginning of job execution, in ten-minute intervals and at the end of the job execution. Jobs generate their resource usage data, which are archived on the file system. The events are logged by each node through a centralized message logging system. The logs are combined and interleaved in time. We evaluate our approach on four weeks of resource usage data (32GB) and rationalized logs (1.2GB). These data were collected for the month of March 2012. It is worth noting that within this time, the system experienced high failure rates making this data sufficient for this analysis. We extracted the 96 elements or counters from the resource usage log as seen in Table 3.2. The size of the time window (tw) chosen was based on two factors: (i) lead time to failure - This is determined by *root cause analysis* of faults and research on Ranger Rationalized logs (ratlog) has shown that the *minimum* lead time to failure of most occurring faults is about 120 *minutes* [25], and (ii) The concept of nodehour [106] was shown to be fine enough to capture erroneous messages. To this end, we set tw = 60 minutes. This time window also allows us to compare our work with Nodeinfo [106], a popular error detection approach. We conducted the experiment on 720 sequences events logs with corresponding resource usage data sequences of which 182 are faulty sequences.

Evaluation Metrics

In measuring the performance of our detection algorithm, we employ the widely used *sensitivity, specificity* and what we called *S-measure* metric. The latter, similar to the known F-measure, is the harmonic mean of *sensitivity* and *specificity*. *Sensitivity*, also called *true positive rate* or *recall*, measures the actual proportion of correctly detected failure sequences to the total number of failure sequences as expressed in Equation 5.8. *Specificity*, or true negative rate, measures the proportion of non-failure sequence which are detected as non-failure

[2] *www.tacc.utexas.edu*

among all non-faulty sequences as seen in Equation 5.9. *S-measure* here is synonymous with the usual *F-measure* in information retrieval, however, in this case, it is the harmonic mean of sensitivity and specificity (see Equation 5.10). Since *sensitivity* or *specificity* cannot be discussed in isolation, *S-measure* combines the two providing us with a balanced detection accuracy.

$$Sensitivity = \frac{TP}{TP + FN} \tag{5.8}$$

$$Specificity = \frac{TN}{FP + TN} \tag{5.9}$$

$$S - measure = 2 * \frac{Sensitivity * Specificity}{Sensitivity + Specificity} \tag{5.10}$$

		Actual Data	
		Faulty	Normal
Detected Result	Positive	TP	FP
	Negative	FN	TN
		Sensitivity	Specificity

Figure 5.6: Evaluation metrics

The parameters TP, FP, TN, and FN denotes *true positives*, *false positives*, *true negatives* and *false negatives* respectively. Figure 5.6 demonstrates the relationship of these parameters. A perfect detection will have sensitivity and specificity value of 1, meaning it can detect the faulty and non-faulty sequences accurately.

Failure Pattern Detection Performance

We aimed to show the detection accuracy of our methodology and then compare our approach with Nodeinfo, a popular error detection approach proposed by Oliner *et al.* [106].

In the experiments, we evaluated our approach under various conditions. Since our approach is based on concepts such as anomaly score and entropy of a sequence, we show the effectiveness of our detection methodology under different values. The aim is to find value combinations where detection accuracy is better achieved, i.e., a high true positive and true negative rate. We observed that a change in mutual information threshold (μ) does not have as much influence on the detection result as do the entropy threshold (φ) and the anomaly threshold (γ) values. Figures 5.7 and 5.8 shows the results of detection with different values of γ and φ. The best detection result is achieved for values of $\varphi = 0.4$ and $\gamma = 0.6$, achieving sensitivity (true positive rate) of about 80% and 78% respectively. This result also demonstrates how the best value of the features which affects detection the most are obtained. Note that the value of sensitivity increases with increase in φ (see Figure 5.7); however, specificity increases with corresponding increase in values of φ. Our approach is able to detect 80% of errors that lead to failures. Further, with the high value of specificity, we conclude that the false negative rate is also low.

It is worth noting here that the detection threshold is not dependent on the system on which the approach is applied. It is dependent on how anomalous the logs and the resource usage data of such systems are.

Since *sensitivity* is a measure of the true positive rate, it is best to detect all faults if possible, and our approach demonstrates that about 80% of faults can be detected. In another sense, it is important that false alarm is reduced (high specificity). From Figure 5.8, we notice that the choice of anomaly threshold (γ) affects detection. False positives tend to increase at values below 0.6 and values greater than 0.6, resulting in reduced *sensitivity*. On the other hand, *specificity* increases with increase in γ.

Results implication: It is better to achieve higher *sensitivity* than *specificity*. This is because it is better to be safe and know that there is no potential failure and deal with the false alarms than having potential failures go undetected. However, it is expected that system faults and eventual failure should be a rare activity, and when false alarm is high, it becomes a disadvantage and may lead to unnecessary attempts towards avoiding failures that never even existed.

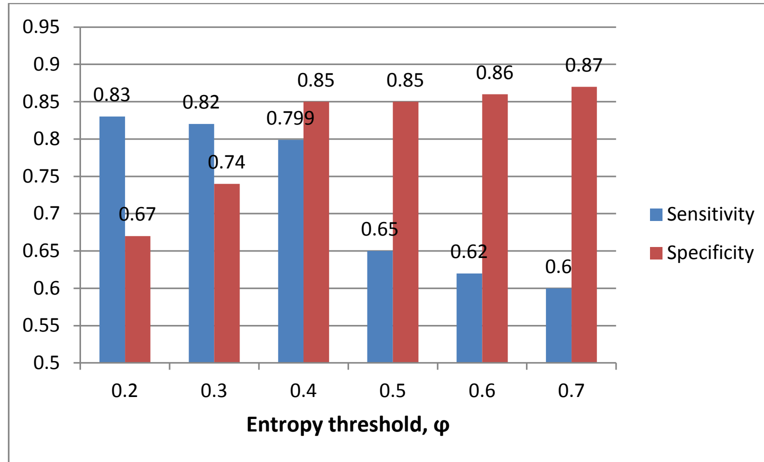


Figure 5.7: Results showing accuracy of our detection approach under varying values of entropy threshold (φ) and $\gamma = 0.6$.

Comparison with Nodeinfo

NodeInfo [106] is motivated by the assumption that similar computers executing similar jobs should produce similar log contents. In that regard, as long as the log lines are tokenizable, the idea can work. It equally leverages the"log.entry" weighing scheme for calculating the entropy of a"nodehour" (nodes within an hour), synonymous to documents in information theory. It first computes the amount of information each token conveys with regards to the node that reported it. Nodeinfo uses Shannon information entropy as defined in [122] to calculate the information by each node. They further obtain and rank "nodehours" according to how high information terms (Nodeinfo) they contain. Nodehours, or what we called sequences of 60 minutes in size, with high Nodeinfo

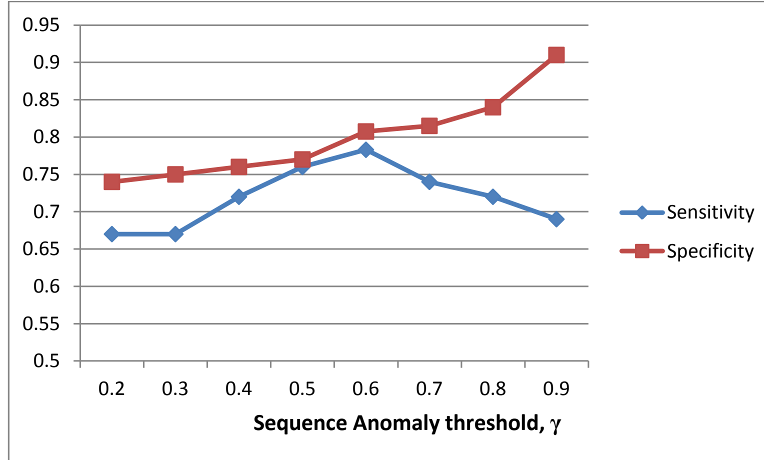


Figure 5.8: Results showing accuracy of our detection approach under varying values of varying anomaly threshold, with $\varphi = 0.4$.

value, is regarded as faulty or containing alerts. We implemented this approach (Nodeinfo [106]) and evaluated in on our event log data. We compare the performance of *Nodeinfo* with our approach. Figure 5.9 shows the detection *S-measure* of both methods. While our method performs consistently better with increase in γ, $Nodeinfo$ consistently decreases with increase Nodeinfo threshold. This means that as we set the informativeness of a sequence to be high, Nodeinfo detects fewer faulty sequences. Our method achieved on average (across all the thresholds used, using S-measure) an improvement of about 50% over *Nodeinfo*. Also, on the best anomaly threshold (0.4) it achieved an improvement of about 30%, that is, our approach can detect an additional 30% of faulty sequence over Nodeinfo. This shows that the use of resource usage as a complement to event logs, as proposed by our approach can be effective in increasing the accuracy of error detection.

Runtime Performance Analysis

We implemented our algorithm in Java and ran it on a system with an Intel i5 (3.10GHz) CPU to evaluate the runtime of the approach. The performance (runtime) of our algorithm is not affected by detection thresholds but only

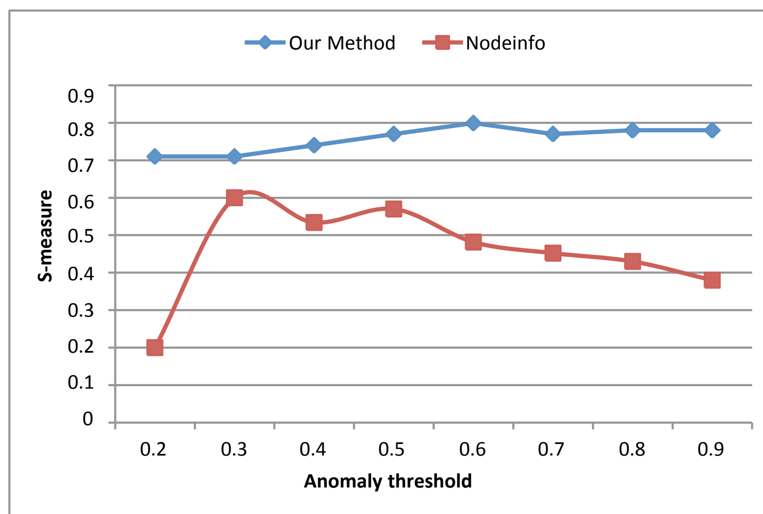


Figure 5.9: Graph showing detection performance (S-measure) of *our method* and *nodeinfo*

by the size of the data. This is mainly during the data transformation and the process of obtaining the anomaly score. The PCA approach to anomaly detection slightly increases the runtime, this is only true if the data is large.

Figure 5.10 demonstrates that the size of data[3] has an impact on the runtime performance. As the size of the logs increases, the runtime also increases; however the increase is not exponential. The increase is gradual, and for a data size of 300MB, the runtime is just about 4 minutes. This is not a challenge as logs may not be this large within a time window; even if it were so, the time taken to process it for detection is reasonably small. The detection process (Algorithm 6) is very fast with just about 2-3 seconds and it is not dependent on the data size at this point. The graph of Figure 5.10 shows the runtime of all the steps involved.

[3]We add both resource usage data and error logs

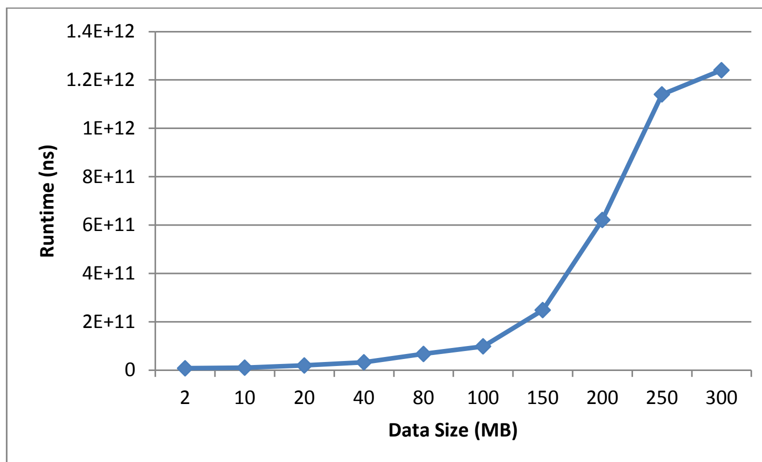


Figure 5.10: Graph showing runtime performance of *our method*

5.3 Detection of Recovery Patterns

5.3.1 Introduction

The faults that occur in cluster systems may or may not eventually lead to a system failure. As rightly observed by Oliner et.al., [102], that in reality, there exists no description of a correct system. This is true since so many components are involved and system administrators are probably unaware of some happenings within the system, sometimes until a problem is noticed. The characterisation of these faults by Gainaru et. al., [43] provides an insight as to how some failures behave. Sometimes, the anticipated failure as a result of these faults and errors may eventually not occur. This happens for network faults if a *recovery* is successfully completed [24], [23]; similarly, *memory errors* may be corrected by error correction codes (ECC), even though, from the events logs, there is every indication that these faults will result in failure [43]. That is, looking at the event logs, the patterns of messages are indicative of impending failure. Most times, it is very difficult for system administrators to know if faults within such time eventually lead to the failure or not. Previous work [53] on detecting faults using logs has shown that this problem contributes to increased

false positives.

We want to identify those sequences which, from their patterns of events are indicative of failure; however they eventually did not end up causing any failure. We call such sequences *recovery* sequences while those that end in failure as *failure* sequences. Obviously, *event logs* or error messages will not provide the needed insight if such sequences end up in failure or not. We propose a novel approach based on change point detection that detects such sequences. To the best of our knowledge, this work presents the first insight into this problem as we have not seen a work that gave it attention. The detection approach demonstrates that resource usage/utilization data can be useful in identifying *recovery sequences.*

5.3.2 Recovery Pattern Detection

To achieve our aim, resource utilization data or usage data is used. We reiterate the point that the resource usage data provides us with an understanding of the happenings within the system regarding how resources are being utilised. For example, a high or low usage of memory or network resources or better still, a sudden change in page swap rate could point to an abnormal behaviour which may lead to failure. In essence, an abnormal use of resources is a pointer to imminent failure.

We conjecture that a system which experienced a *successful recovery* from network error or memory error corrected by ECC may behave differently in its resource usage. Even though error messages may not provide a clear indication that failure will eventually occur, resource usage data within such time could provide a hint. Usage data within such time window could show unusual use/utilization, however, it may eventually recovered successfully and not end in failure. Normal behaviour towards time of expected failure could point to successful recovery.

In this section, we detail steps taken to identify sequences with successful recovery from faults by detecting points of unusual or abnormal change within

the sequence of resource utilization data for which failure is expected or has occurred. We utilised the idea of change point detection (CPD) to perform this. Before the technique could be applied however, the data must first be transformed to a format that can be used easily by the algorithm.

Data Preparation

Resource usage data as earlier explained, contains how much resources are used on a particular node as captured by different resource counters (see Table 3.2). Each counter captures the amount of resources they are associated with. For example, a network counter (*rx_mgs_dropped*) captures the amount of messages dropped by a particular node.

Hence a line of logged usage data contains all the counters and their usage values captured within a certain period. Let us call these lines of logged usage data as events, e_i. These events are streams of time series data. For the purpose of our research, we capture these events within given time window, t_w, called subsequence, x_i. A sequence, $S = x_1, x_2, ..., x_n$, is then a stream of subsequences as illustrated in Figure 5.11. It is worth noting here that the choice of t_w may be dependent on the time to failure of a fault and component. A reasonably small time is chosen to avoid capturing different usage patterns within a subsequence; however, the time should be large enough such that the subsequence is still informative.

$$S = \{\underbrace{e_1, e_2}_{x_1}, \underbrace{e_3, e_4}_{x_2}, \ldots, \underbrace{e_{n-2}, e_{n-1}, e_n}_{x_n}\}$$

Figure 5.11: Sequence of resource usage data

We extract each x_i as a vector of the sum of resource usage for each counter. For example, given subsequence x_1, with counter *tx_bytes:* 267, *read_bytes:* 302, etc, then the vector x_1=[267 302 ...]. Hence, the amount of resources used on nodes n_i in subsequence x_i are summed up for each counter. These values are then scaled to values between 0 and 1, forming a probability distribution. This

is because the change point detection algorithm accepts the data as a probability distribution. It also makes the data easier to handle and explained. We then construct a matrix of the sequence where the subsequences forms row vectors. Hence, given k number of counters and n subsequences, then the matrix M is as given in Figure 5.12.

$$M = \begin{pmatrix} x_{1,1} & x_{1,2} & . & . & . & x_{1,k} \\ x_{2,1} & & & & & \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ x_{n,1} & x_{n,2} & . & . & . & x_{n,k} \end{pmatrix}$$

Figure 5.12: Data matrix M with n subsequences of S, where $x_{n,k}$ is the value of counter k in subsequence n.

From the matrix M of Figure 5.12, a vector representing S is formed by summing the values of each counter in a subsequence divided by the number of counters. That is, subsequence $x_1 = x_{i,1} + x_{i,2} + ... + x_{i,k}$ divided by k (counter size). This is done for all the n subsequences. Hence, the vector forms the input to the detection algorithm.

Change Point Detection

The objective of anomaly detection is to find a data point or data points that *behaves differently*. The anomalousness now depends on the field of application where an anomaly is a rare behaviour. Change Point Detection (CPD) [92, 130] is an anomaly detection method where it detects *"drastic change"* observed from a sequence distribution. These points of drastic change are possible anomalies. Two classes of CPD are commonly used depending on the problem; they are: *Real-time change-point detection* and *Retrospective change-point detection* [92]. The former deals with detecting real-time changes in applications, a good example is responses in robots. The later deals with applications with longer response times and it is also used to deal with retrospective data.

In this work, we employ retrospective CPD to detect sudden changes in the

utilisation of resources by a supercomputer system. Such sudden change could point to abnormal behaviour in the system. For example, a sudden peak in memory or network resource usage could signal the presence of faults and/or errors.

The reason for the choice of CPD approach in this study is connected to the nature of our data. The resource usage data are collected and logged as streams of time series data which is formed by the probability distributions of the resources used on a node by running jobs. Therefore, the level at which resources are utilised may vary with time and this changes can be captured using CPD. This is our motivation for using change point detection. We will discuss briefly two Retrospective CPD approaches we employed in this work.

Cumulative Sum Change-Point Detection: Cumulative Sum (CuSUM) CPD approach [130, 131], is based on the fact that a sudden change in parameter value corresponds to a change in the expected value of log-likelihood ratio. From the name, it tracks the cumulative sums of the differences between the values and the average. At points where the values are above average, the cumulative sum steadily increases. Therefore this method involves finding the mean and its difference with observation values.

Given series of data $S = x_1, x_2, \ldots, x_n$, we first initialise the cumulative sum, $cS_0 = 0$ and obtain the mean of S (the row vectors), given as $\bar{x}$,

$$cS_{i+1} = cS_i + (x_{i+1} - \bar{x}) \tag{5.11}$$

for all $i = 1...n$.

Abrupt change points are those points with cS_i values above a certain threshold value.

Divergence-Based Dissimilarity Measure: In this approach, a dissimilarity measure is introduced. We used Kullback Divergence (KLD) measure [92].

The Kullback Divergence of two sequence distributions x and y (*for simplic-*

ity, we assume $x = x_i$ and $y = x_{i+1}$) is given by:

$$KLD(x\|y) = \sum_i x(i)log\frac{x(i)}{y(i)} \tag{5.12}$$

where i is the index of probability values of vectors x and y.

Figure 5.13 shows the CPD charts of sequences that end in failure and one that recover. It can be noticed that usage observation values decreases for the *recovery* sequences as time progresses. It is lower for the recovery compared to failure sequence. This gives an indication into how these two sequences could behave based on how the resources are being used (during failure, recovery). We utilised this for our detection algorithm explained in the next section.

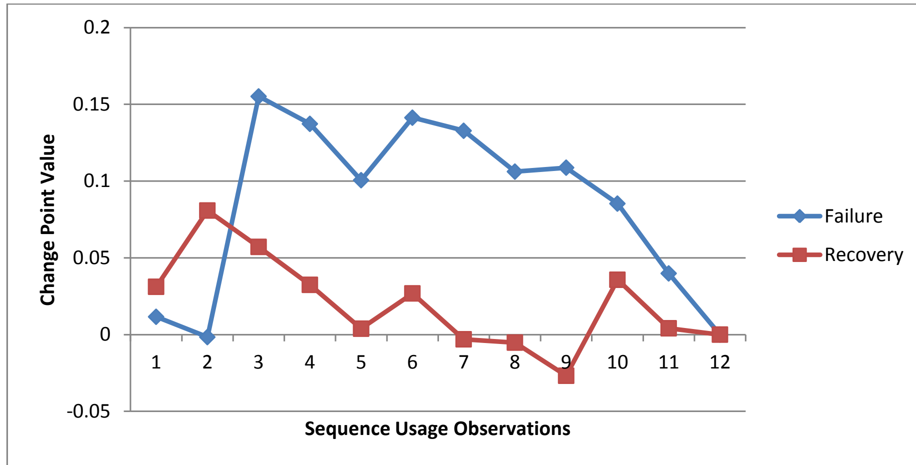


Figure 5.13: Graph showing the change point behaviours of both recovery and failure sequences

Detection

We detect multiple change points within a failure sequence. We conjecture that sequences that eventually end in failure are likely to contain change points and/or a sustain presence of such points leading up to time of failure. Meanwhile, sequences that eventually experience recovery may not contain many change points or sustained change points leading up to expected time of failure.

These sequences may be characterised by relatively normal resource utilization if there is a successful recovery from faults. Given the observations of usage data x_i within a sequence S, that is, $S = x_1, x_2, ..., x_n$, where observations, x_i, are made within a certain time interval and the time of occurrence of x_i, $t_{x_1} < t_{x_2} < ... < t_{x_n}$, then we conjecture that for any failure sequence S, an abnormal use of the resources can be noticed throughout the sequence and it is likely to be sustained across t_{x_1} to t_{x_n}. However, a recovery sequence will likely experience normal behaviour or normal resource usage as the time approaches failure. This implies that it may likely contain less change points as it approaches the expected failure time.

Algorithm 7 Recovery sequence detection

```
 1: procedure DETECT(S, th)                        ▷ th is the detection threshold
 2:     cp = null                       ▷ keeps the list of points above threshold
 3:     x_i ∈ S                                 ▷ vectors (subsequences) of S
 4:     for  i = 1 to |S| − 1 do
 5:         p(x_i) =CPD(x_i, x_{i+1})      ▷ CPD represents either KLD or CuSUM.
 6:         if (p(x_i) >= th then                  ▷ th is change point threshold
 7:             add point (i) to list of change points (cp)
 8:         end if
 9:     end for
10:     if (if there are more than a point i greater than midpoint) then
11:         return Failure
12:     else
13:         return Recovery
14:     end if
15: end procedure
```

From Algorithm 7, we detect multiple change points within the sequence. We keep the points which are seen as change points for the sequence. The sequence with change points occurring beyond the midpoint of the sequence will likely end in failure as earlier explained.

5.3.3 Results

Our aim is to develop a methodology to detect sequences that recovered from faults and did not end in failure. These sequences contribute to false positives on the earlier failure detection approach since they they equally produce event

messages indicative of failure. To achieve this aim, we utilized the *resource usage data* of the Ranger supercomputer and not the error logs. The approach is then evaluated by conducting experiments on the resource usage data.

As earlier explained in Chapter 3, the resource usage data were collected using TACC_Stats [59] that takes snapshots of utilization data of the 96 counters. The snapshots are taken within ten-minute intervals . Jobs generate their resource usage data on a particular node, which are then logged to the file system. The data is logged by each node through a centralized message logging system. The logs are combined and interleaved in time.

From the data, more failures actually took place within the first and second weeks of March 2012 with few occurring in the third and fourth weeks. Among these failure sequences are those that eventually did not end up in failure, but experienced a *recovery*. We had a total of 720 sequences of which 182 are real failure sequences and 72 *recovery* sequences.

In the experiments, we evaluate our approach under various detection threshold values. The values of detection threshold, th is varied to obtain better values for both sensitivity and specificity. We show results and discuss the two CPD methods (CuSUM, KLD) used.

From the results seen in Figure 5.14 for using CuSUM approach, the true positive rate (sensitivity) performs poorly at $th = 0.1, 0.2$. It consistently increased (maximum sensitivity of about 90%) as the value of th is increased. This shows that the more we increase the value th, the better the detection of the recovery sequences. It achieved highest sensitivity at $th = 0.7$, which remains constant for higher values of th. Likewise, the specificity is highest at lower values of th as expected and reduces from 70% to 20% at $th = 0.6$ and remains so for higher values. These results demonstrate that we can achieve good detection of recovery sequences when we use CuSUM change point detection method. For this approach (CuSUM), a better result (S-measure) is obtained at threshold $th = 0.3$ as seen in Figure 5.10. It achieved about 63% detection. However, the high false positive and false negative rates renders this approach less effective.

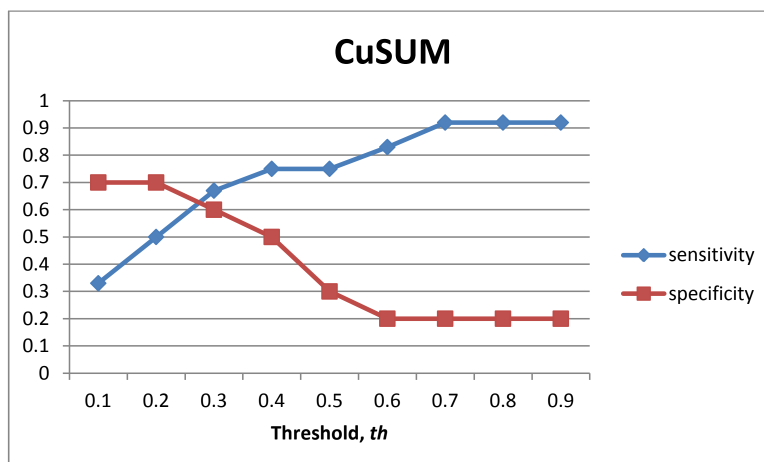


Figure 5.14: Result showing accuracy of detecting recovery sequences among failure sequences using *Cumulative Sum* change point detection and varying values of detection threshold, *th*

.

Similarly, using KLD method (see Figure 5.15), the results are similar to CuSUM. A highest sensitivity of about 84% is observed when $th = 0.5$ and more. The specificity on the other hand, decreases with increase in th. The lowest specificity (10%) is obtained at $th = 0.6$ and remained so for higher values. Comparatively, CuSUM seems to slightly perform better over all the thresholds used. However, looking at the *S-measure* in Figure 5.16, KLD performed high with detection of 64% at $th = 0.2$. This result is almost similar with the CuSUM approach (1% difference); the only difference is that they are achieved at different detection thresholds.

Based on these results, it is very possible to achieve good detection of sequences which did not end in failure (*recovery sequences*) from usage data. Even though, this is not the best performance expected, however, it is a good starting point for exploring the use of resource utilisation data of cluster systems to detect both failure sequences and recovery sequences applying change point detection method. One main challenge with our method is the insufficiency of data, that is the number of recovery and failure sequences. We believe a better result can be achieved if more data containing identified failures and recovery

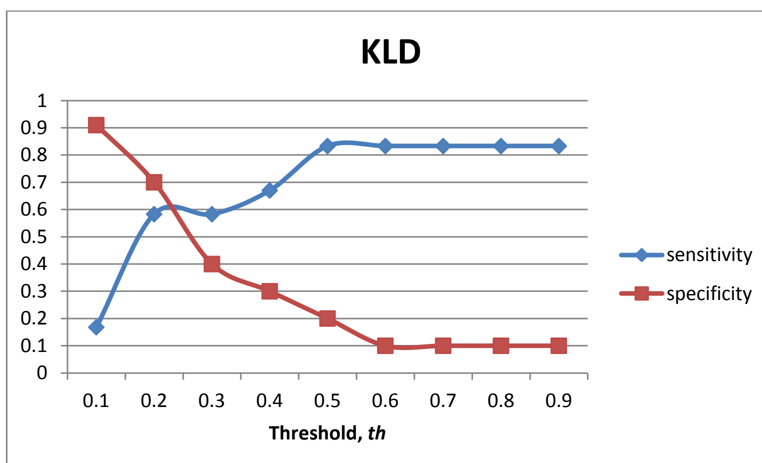


Figure 5.15: Results showing accuracy of detecting recovery sequences among failure sequences using *KLD* change point detection, and varying values of detection threshold, *th*.

sequences are available. The four weeks data we used, as earlier mentioned, experienced more failure and recovery within the first two weeks of cluster system operation and very few at weeks three and four. We speculate here that this approach may not be the best for faults with no impact on resource utilization. This is because it depends fully on the how the resources are used.

5.4 Improving Failure Pattern Detection

The performance of the failure pattern detection discussed in previous sections can be improved by combining the detection approach with detecting recovery patterns. The idea is that *recovery patterns* if detected can reduce the number of false positives. This is because most times, the recovery patterns behave similar to failure patterns. Hence, it explains the poor *specificity* values of the detection algorithm.

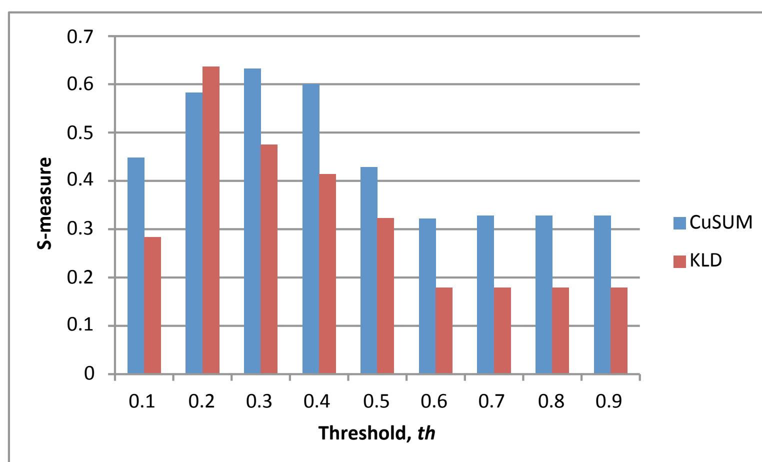


Figure 5.16: Graph showing detection performance (S-measure) of both CPD methods used

5.4.1 PCA and CPD Failure Detection Algorithm

We have shown earlier that these *recovery patterns* can be identified with about 65% detection accuracy[4]. The motivation here is that by combining the PCA-failure detection approach and the CPD-recovery detection methods, we could improve the general performance of the failure detection.

The detection based on the CPD and the PCA discussed earlier is performed as follows: A pattern detected as *failure* by the PCA-detection seen in Algorithm 5 is only a failure if the same pattern is *not* a recovery pattern based on CPD Algorithm 7 discussed. This step stems from the fact that *recovery* patterns are non-failure patterns, and these patterns can easily be detected as failure patterns due to the nature of the events logged (which are most times indicative of impending failure). This approach is aimed at reducing the detection *false positives* due to the recovery patterns. It is expected that the combined PCA and CPD failure detection algorithm will bring about improved identification of non-failure patterns.

[4]by detection accuracy here, we mean detection sensitivity

5.4.2 Results

Similar to the experiment in Section 5.2.5, we conducted the experiment and maintained the evaluation metrics. The value for CPD threshold is chosen as $th = 0.2$, as this is the value for which recovery detection was highest (see Figure 5.16).

From the results of Figure 5.17, the *sensitivity* decreases with increase in the anomaly threshold (γ) values. On the other hand, *specificity*, as expected, increases with increase in γ. The highest value of *sensitivity* is achieved by combining both PCA and CPD-detection is about 71% at $\gamma = 0.4$. With increase in detection threshold[5], more faulty patterns are detected as normal patterns (false negative increases). This implies that the anomalousness of a pattern doesn't have to be very high for it to lead to failure. In essence, a system that demonstrates above 50% anomalousness has about 71% chance of resulting in a failure.

The CPD part of the algorithm is meant to reduce the false positives that arise from the algorithm. Most times, these patterns contain events that indicate the presence of failure, however the system experiences recovery. In particular, the CPD is aimed at improving *specificity* which increases with increase in γ, as seen Figure 5.17.

Comparing the performance of this approach (PCA and CPD) with detection explained in previous sections (based on PCA-anomaly, event entropy and mutual information), we can clearly see that the latter performs better. This can be explained as from CPD-recovery detection, the performance was about 65%; with 35% wrongly detected recovery patterns. This contributes to the reduced performance when the two methods are combined. Hence we conclude that, unless a better performance is achieved in detecting recovery patterns, the combination of PCA and CPD for detection does not help.

[5] when the resource usage deviation from normal is reasonably high before we suspect an anomalous behaviour in the system

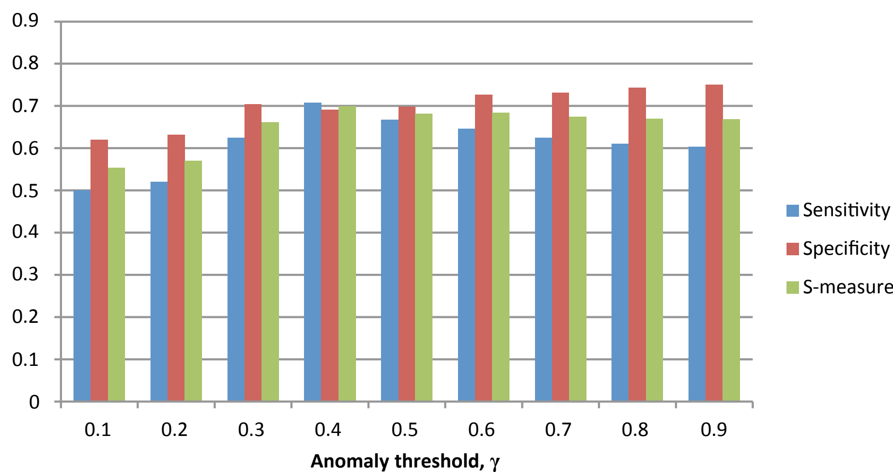


Figure 5.17: Graph showing detection performance of combining detection based on *PCA-anomaly* and *CPD-Recovery*.

5.5 Summary

In summary, this chapter's main and *novel* contributions are as follows:

- We develop an algorithm that uses the console logs to detect erroneous runs. The algorithm achieves this by clustering together nodes that exhibit similar behaviour, through the use of the mutual information and entropy concepts.
- Usage log data provide an understanding of cluster systems resource utilization. Abnormally high utilization may suggest an impending failure due to some errors in the system. To be able to capture the anomalous behaviour of jobs running within the system, we employ an unsupervised detection approach, PCA, to do this. Highly anomalous jobs present within a sequence signify the presence of faults which could lead to failure.
- Lastly, we utilised both the patterns mutual information and entropy from event logs and the outlier/anomaly level of sequences from its resource usage data to detect if a sequence is likely to lead to failure or not. We

propose a detection algorithm based on these systems information from both event logs and resource usage data. We then compare our method with an existing approach.

Our approach is very promising: it is able to detect faulty sequences with high accuracy and, when compared to the well-known Nodeinfo approach [106], it outperforms it greatly.

Relation to other chapters: This chapter has shown that errors can be detected with high accuracy. The question that follows is: can failure prevention or avoidance or error handling methods like *checkpointing* be successfully completed before an impending failure is experienced? This question is valid since preventive checkpointing[6] is said be a promising approach in the exascale computing era [16]: hence there is a need to extend the error handling time if possible. The next chapter addresses this problem.

[6]Triggered whenever an error is detected

CHAPTER 6

Early Error Detection for Increasing the Error Handling Time Window

6.1 Introduction

HPC systems' failure is a recurrent event owing to its scale and complexity. Given that dependability is an important property for such systems, the ability to properly handle (fail-stop) errors is crucial. One of the fundamental approaches to handle such failures makes use of the *checkpoint-restart* (C/R) technique, where the application state is saved at regular interval during execution (i.e., checkpoint) and restarted at the latest checkpoint, in case of a failure. However, the C/R technique faces one major challenge: the checkpoint time is significant, roughly currently in the order of $25 - 30$ minutes [14, 16]. However, it is predicted that, on very large scale platforms (e.g., exascale platforms), with expected Mean Time To Interrupt predicted to be very low (one hour or less), little progress can be made due to the significant proportion of time devoted to C/R [11, 14]. A possible alternative to checkpointing is task migration but one of the limitations of this technique is that checkpointing is again required to deal with false negatives (i.e., when the system does not detect an impending failure) [16], making C/R a cornerstone for building resilience into HPC systems. Another approach is to combine proactive checkpointing (where the state of a given node is saved) with preventive checkpointing(the state of the entire system is saved), supported by accurate failure prediction [11].

Thus, to address the checkpointing (or error handling) time, there are several research avenues being pursued. One way of reducing the C/R proportion is to reduce the size of the checkpoint. With little data being available regarding the

typical size of a checkpoint, it is widely believed that programmers save more data than required as they cannot easily track data structures updates between checkpoints. Thus, techniques such as memory exclusion and compiler analysis to detect dead variables have been proposed. Another alternative to reduce checkpointing time is to reduce the usage of disks to store checkpoints [16].

However, one technique which holds promise is *preventive checkpointing* [16]: *Error detection* (or *failure prediction*) is used to trigger a checkpoint before the error propagates through the system. To enable this technique, two important challenges exist: (i) developing efficient error detection techniques (to trigger checkpointing) and (ii) the time window between the time of error detection and the actual failure needs to be large enough (to allow for checkpointing to complete) and also for the system to make progress. With the size of RAM memory expected to increase, techniques need to be developed to keep this time window as *large* as possible. We call this time window the *error handling time window*, which is the focus of this paper.

Detecting fault symptoms (i.e., errors) using error logs has received good attention, with reasonably good results, e.g., [9, 43, 112, 147]. Most of these approaches have attempted to identify individual faulty events within the data. However, in many cases, individual events may not be sufficient to indicate an impending system failure. Other approaches for error detection within patterns or faulty patterns/sequence in the recent past have used the concept of log entropy of the log messages seen within the sequences [93, 96, 106]. Log entropy generally leverages the inherent changes in the frequency of event patterns to capture the behaviour of a system. Oliner *et al.* [106] attempted to capture the sequence information, or what they called Nodeinfo, of a nodehour, an area within which the log can be considered as faulty. A recent approach combines both entropy and the concept of mutual information of patterns to discriminate between faulty and non-faulty patterns [53].

6.1.1 Motivation

The systems executes resource-intensive applications such as scientific applications which require the architecture or the system to display a very high level of dependability to mitigate the impact of faults. As earlier stated, typical error handling techniques such as checkpointing and task migrations are expensive as they incur a high completion latency. To ensure the success of such techniques, enough time should be allowed for these to complete, i.e., the error handling time should be greater than the completion latency of the error handling mechanisms. However, error detection based on event logs may leave a short error handling time window insufficient for such mechanisms to complete. Also, the log files are known to be *incomplete*, i.e., the log entries do not provide sufficient information regarding the behaviour of HPC systems that can be used to accurately detect errors in the system and *redundant*, i.e., several log entries may be due to the same event. The logs may contain several error events that relate to the same fault. These issues make error prediction or failure prediction based on error log files challenging, leaving a small *error handling time window*.

6.1.2 Problem Statement

In this section, we explain the problem we address in this paper, and the challenges associated with it.

Log messages that are error messages capture the nature of the underlying problems in the system. However, due to the incompleteness nature of event logs, a single error message may not accurately predict an impending failure. For example, a loaded network may cause a network timeout. However, this single event may be later followed by a retransmission after which, a successful recovery is achieved. Thus, to increase the accuracy[1] of failure prediction, the behaviour of the system needs to be observed over a period of time. It is stipulated that some failures can be predicted with more than 60% accuracy [14]. Because of the high overhead associated with false positives, the prediction of

[1] Accuracy here is the ability to detect actual failures while also reducing false positives

a failure based on the observation of an error message is not advised. Rather, to keep high accuracy while minimizing false positives entails observation of an erroneous event sequence for failure prediction. Thus, the problem is to develop a methodology such that (i) accurate failure prediction is high and (ii) the failure lead time is high, enabling the system to make sustained progress even after checkpointing is done (or enough time for checkpointing to complete).

We now pose the main problem we address in this paper as follows: Given an event log file F, and possibly a set of other log files $F_1, \ldots, F_n$, a time window W_e for event log analysis, and a time window W_i for every other log file F_i, develop a methodology such that the time window between the time of failure prediction and the time of system failure is high. We call this time window the *error handling time window*, as this is the amount of time the system has to tolerate the fault, i.e., correct the error, before failure occurs.

6.1.3 Objectives of the Chapter

To complement error log files, this chapter propose the use of *resource usage information* to aid in early error detection. Particularly, it seeks to detect errors early with high probability so as to *increase* the error handling time window so that recovery procedures such as C/R can successfully complete [54]. To increase the time window, we propose a *novel* approach based on the observation that anomalous resource usage could lead to system failure [25]. Thus, in this chapter, we seek to develop a methodology that combines both resource usage data and error logs to obtain a larger time window to support error handling.

The remainder of the chapter is structured as follows: The methodology or the steps taken towards solving the problem is discussed in Section 6.2, we presented the case study and results in Section 6.3. We briefly summarise the chapter's contributions in Sections 6.4.

6.2 Methodology

Our approach thus proposed to use another log file, namely *resource usage data*, which reports the amount of resources that are being utilised by each job on a node. Inconsistent or anomalous amount of resource usage by jobs on a particular node could point to potential problem in the system, that can eventually lead to system failure [25][2].

In this section, we detail our methodology, based on both event logs and resource usage logs, for obtaining an increased error handling time window. The workflow starts with the time of the failure as input, which we denote by T_f. We first conduct (i) a *root-cause analysis* on the event logs to obtain the time T_e at which the root-cause event of the failure occurred, with $T_e < T_f$. (ii) We then run an *anomaly detection algorithm* on the resource usage data to identify the job(s) with anomalous resource usage data. The first time this anomaly occurs, prior to T_e, within a given time window W_r is noted. We denote this time by T_r. We further attempt to push back the failure prediction time through (iii) the detection of *change points*. The rationale behind this is that when a resource usage anomaly occurs, this anomaly cannot happen in one step. Rather, it happens in a few steps. For example, if the workload suddenly increases, the job may request more and more resources (over a certain time) rather than the request being serviced in one go. Thus, the observation of change points may indicate impending failure. We now explain the main techniques involved in our methodology as depicted in Figure 6.1.

6.2.1 Root Cause Analysis

Event logs of large cluster systems are made up of streams of interleaved events. This is because of the high degree of interactions among the system hardware and software components. Most times, only a small fraction of the events over a small time span are relevant or could point to pending failure, i.e., only a small

[2] Observe that this may not always be the case. For example, a job starting will suddenly acquire a relatively large amount of resources to run, though this is not anomalous.

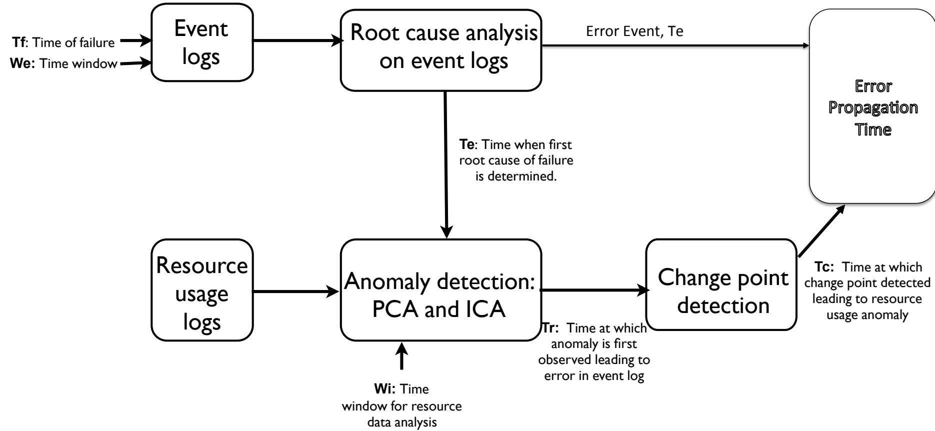


Figure 6.1: Methodology work flow

proportion of these events are error events. Identifying the causes of system failures is necessary and event logs being a source of health information, system administrators have to identify the small useful fraction of the events that are related to the failure. Such an activity is typically very challenging. In order to trace the likely root-causes of failure in logs, we identify events that are highly correlated with frequently occurring failures. We use an established fault diagnostic tool called FDiag [23, 24] for this purpose. It extracts from large event logs of cluster systems, error events which are regarded as causing system failures. These events are those which are highly correlated with the failure events. Given a time window W_e before the failure, we seek root-causes within W_e. FDiag basically works as follows: Given a known system failure, FDiag identifies patterns of system events which occur across the nodes of the cluster system, extracts only the correlated events (with the failure). These pattern of events identified are within given time period, providing the fault events and the time for which they occur. FDiag [24] employ statistical correlation approach to obtain these faults which causes recurrent failure.

6.2.2 Anomaly Detection

An impending failure in a cluster system is manifested in event logs but sometimes are not logged early enough before the failure occurs. The behaviours which are highly correlated with failures may not manifest early in the logs; in other words, the sequence of errors are reported too late especially for prediction purposes. On the other hand, resource utilization data or usage data records how the resources are being used by the running jobs on each node. These usage data can record early abnormal behaviour experienced by each job on a particular node [40]. It has been shown that a correlation exists between anomalous resource usage and system failures [25] and Gabel *et al.* [40] has demonstrated that counters can indeed capture latent faults and that these can be detected. In this section, we explain our approach for extracting anomalous running jobs, which are conjectured to be indicators of a problem within the cluster system, before these problems manifest themselves in the event logs. We also extract the time for which anomalous behaviour is experienced. However, for a given root-cause, there may be several preceding resource usage anomaly, we impose a time window W_r preceding the root-cause, during which we take the time at which the first anomaly is noted. We denote this time by T_r. We first have to transform the resource usage data into a form suitable for analysis, before explaining the anomaly detection algorithms used in this paper.

Data Transformation

The resource usage data contains records of each jobs running on a particular node with values for its usage as carried by the elements/counters. In order to appropriately capture the behaviour of the system with these nodes and it running jobs, there is need to transform this data into a format (matrix) that can be used easily. The data which captures how much resources is being used inherently describes the state of the system within a given time window. As earlier said, abnormal resource usage could point to potential system failure [25]. In order to detect these abnormalities in resource usage, we properly extract the

counters or attributes/elements, where the state of the resources for each job on each node within given time are captured. Hence, from the resource usage data, the time window used is dependent on the time of event fault. Given the time at which first fault is observed from the *event logs* is T_e (*from root cause analysis of fault in previous section)*, then, time, T_x, under which the resource usage data will be considered is $T_x = T_e - t_w$, where t_w is the time at which we begin to observe anomalous behaviour.

A *Resource Usage Feature matrix* is formed, where the features are the jobs running on each node within time T_x. For any k counters and n running jobs, we construct a $n \times k$ feature matrix $J = [j_1, j_2, ..., j_n]^T \in \Re^{n \times k}$ with columns representing the counters or the elements of the resource usage and the row vector $j_i = [j_{i,1}, j_{i,2}, ..j_{i,k}]$, $i = (1, 2, ..., n)$ representing the different running jobs on each node. Figure 6.2 describes the matrix.

$$J = \begin{pmatrix} j_{1,1} & j_{1,2} & . & . & . & j_{1,k} \\ j_{2,1} & & & & & \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ j_{n,1} & j_{n,2} & . & . & . & j_{n,k} \end{pmatrix}$$

Figure 6.2: Data matrix J with i features, where $j_{n,k}$ is the value of counter k by job n.

The intersection of row vector value and column vector, j_{ik} represents the resource usage for job i by counter/element k. These resource usage values are summed for each similar counter within same time window and the same job. Specifically, j_{ik} = usage values of counter k reported by job i. This basically captures each jobs resource usage on a particular node.

Anomalous Job Extraction

The resource usage data contains information about how the resources resources (e.g. CPU, I/O transfer rates, virtual memory utilization) of each of these nodes by each job running in the cluster system are utilised or used. It provides

useful hint regarding unusual behaviour of given jobs in terms of its rate of resource utilization within given time. In order detect unusual jobs, we use the transformed resource usage data features as seen in matrix of Figure 6.2 as input data to our detection algorithm. This section aim at obtaining anomalous jobs which could significantly point to problem(s) in the cluster as observed by the anomaly score of the jobs within a period.

We introduce an unsupervised approach to detecting anomalous jobs based on principal component analysis (PCA) [84] and independent component analysis (ICA). Both have been a widely and efficiently used feature extraction method in different fields and also for identifying anomalous node behaviour [80]. However, both methods can reveal the inner structure of data and are suitable in explaining the variance. Other anomaly detection methods, such as one-class SVM [72], are supervised approaches that require the data to be labelled, which is expensive. PCA has been previously shown to be effective in detecting faults in systems using console logs [147], hence our reason for employing both PCA and ICA in this work. We utilise these methods to find anomalous jobs from resource usage data.

Principal Component Analysis (PCA)

PCA is utilised in the chapter in similar way as explained in previous chapter; where, given $J = [{j_1}^T, {j_2}^T, ..., {j_n}^T] \in \Re^{n \times k}$, and each row j_i representing $k-dimension$ data instances of jobs, the v-dominant eigenvectors are obtained.

Independent Component Analysis, (ICA)

ICA [66] is a technique used to find linear representation of variables, measurements to reveal their independence. This involves revealing the linear transformation that maximises the statistical independence between its components. It is a generative model that assumes that components are statistically independent and these components are also assumed to be non-gaussian in distribution.

We adopt the FastICA [66] algorithm implementation of the ICA, for its

fast and efficient performance. The process involved in performing ICA is summarised as follows: A *centering* of the data is performed. This is a pre-processing step for ICA that involves making the random vector a zero-mean variable by subtracting its mean vector. The next preprocessing step is *whitening*. In this step, the observed vector variables, x, are linearly transformed into a new vector $\bar{x}$ with uncorrelated components with unit variance, that is, $\bar{x}\bar{x}^T = I$. This step is done using eigen-value decomposition (EVD) of the co-variance matrix C. The non-zero eigenvalues, λ_i of the covariance matrix, C, are obtained . Let $J = diag(\lambda_1, \lambda_2, ..., \lambda_k)$, $E = [e_1, e_2, ..., e_k]$, then a whitened vector $\bar{x} = EV^{-\frac{1}{2}}E^T x$.

Recent and detail advances on ICA can be found in [65].

Anomalous Jobs Detection in Usage logs Using PCA or ICA

We utilize both PCA and ICA to identify anomalous jobs running in a cluster system within a given time window from the resource usage data features. The approach is based on an assumption that, *for every data point, removing or adding it contributes to the behaviour of the most dominant direction.* This is because PCA/ICA relies on calculating the mean and the data covariance matrix in obtaining eigenvectors, and it is observed to be sensitive to the presence of an outlier. Hence, in this approach, the *outlierness* of a job can be determined by the variation in the dominant principal direction. Specifically, by removing an outlier point or job, the direction of dominant principal component changes however a normal data point will not change it.

So, given data points $J = [{j_1}^T, {j_2}^T, ..., {j_n}^T]$, the leading principal direction d from J is extracted. Then, for each data point j_i, we obtain the leading principal component, d_i, of J without j_i. We use *cosine similarity* to compute the outlierness a_i of point j_i, which is the dissimilarity between d and j_i. The algorithm presented in Algorithm 8 is used to compute the set of anomalous jobs. A point is considered anomalous when its "outlierness" is greater than a given threshold γ.

The same algorithm follows for the ICA detection approach, where the lead-

ing independent component is computed instead of the leading principal component.

Algorithm 8 Extracting anomalous jobs and time from usage data

```
1: procedure AnomalousJob(J, γ)
2:     d = Leading principal/independent direction of J
3:     k = 0
4:     aList                                        ▷ keep list of anomalous jobs
5:     atime                                        ▷ time of first anomaly
6:     for each j_i in J do
7:         d_i = Leading principal/independent direction of J without data
   point j_i.
8:         sim(d_i, d) = (d_i · d) / (||d_i|| ||d||)
9:         a_i = 1 − |sim(d_i, d)|
10:        if (a_i ≥ γ) then
11:            aList.Add(j_i)
12:            aTime = time(j_i)    ▷ keep only the earliest time when anomaly
   occur
13:            k + +
14:        end if
15:    end for
16:    if (k > 0) then
17:        Return aList and aTime          ▷ return all anomalous jobs and the
   earliest time
18:    else
19:        Return No anomalous jobs
20:    end if
21: end procedure
```

An example of the output of Algorithm 5 is shown in Figure 6.3, after running Algorithm 8 with PCA on the resource usage data for the Ranger supercomputer. The points in red indicate the likely anomalous jobs. Figure 6.4 shows the anomalous jobs after running Algorithm 8 with ICA on the same dataset.

Obtaining Counter Relationships

In this section, we describe our approach to uncover possible relationships that exist among *resource counters*. These relationships can indicate that particular resources are the causes of an anomaly. For example, does a high *mmap* counter value have any correlation with the *dirty_page_hits* counter and if so, what does it mean to the behaviour of running jobs in the cluster, or can it point to any

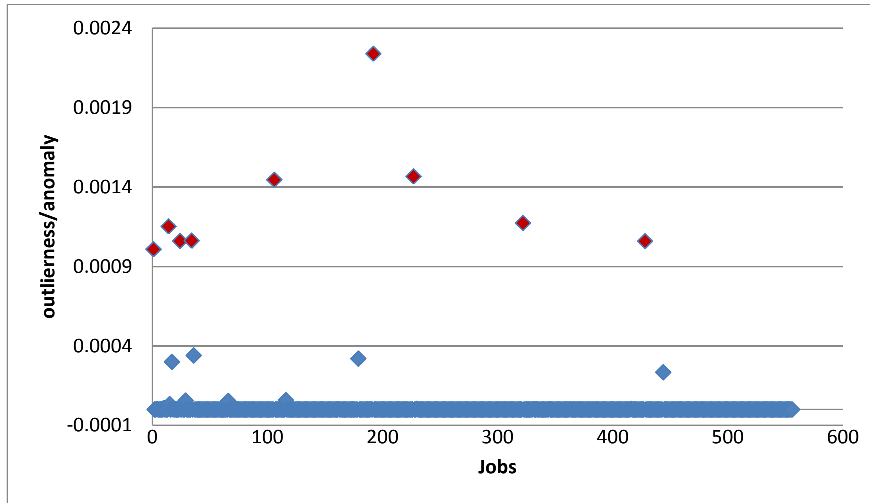


Figure 6.3: Distribution of jobs outlierness of a sequence using PCA

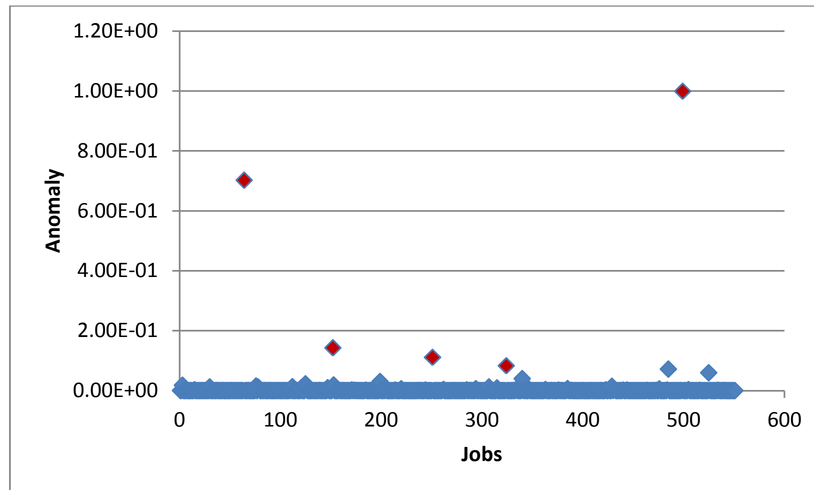


Figure 6.4: Distribution of jobs outlierness of a sequence using ICA

normal or abnormal behaviour? This section aimed to look at the categories of counters (*network, memory or storage*).

Any unusual behaviour of the resource counters can be detected using PCA or ICA anomaly detection. We first employ the PCA approach to obtain anomalous counters. The behaviour of the counters is unclear, with most of them behaving the same (hidden relationships). This can be attributed to the fact that PCA assumes a Gaussian distribution of the data. Hence, in order to understand the relationships between the counters, in this case, we employ the idea of Maximal Information Coefficient (MIC) by Reshef et al., [113]. MIC is promising and it has not been used in this area before.

MIC is a measure of dependence for two-variable relationships. It calculates the measure of dependence for each pair, rank them and further examine their top scoring pairs. If a relationship exists between two variables, then a grid can be drawn on the scatterplot of the variables that partitions the data encapsulating the very relationship. Hence, all grids up to the maximal grid resolution are explored; computing the largest possible Mutual Information (MI) for every pair of integers (n, k) dependent on the sample size. The values of these MIs are further normalised to values between 0 and 1 to enforce fair comparison between grids of different dimensions. A characteristic matrix J is defined as $J = (j_{n,k})$, where $j_{n,k}$ is the highest normalised MI achieved by n-by-k grid, and MIC is the maximum value of M.

Formally, for a grid G, let I_G denote the mutual information of the probability distribution induced on the boxes of G, where the probability of a box is proportional to the number of data points falling inside the box. Then the characteristic matrix, $J_{n,k}$ as shown below.

$$J_{n,k} = max \frac{I_G}{log(min(n, k))} \tag{6.1}$$

MIC is the maximum of $J_{n,k}$ over ordered pairs (n, k).

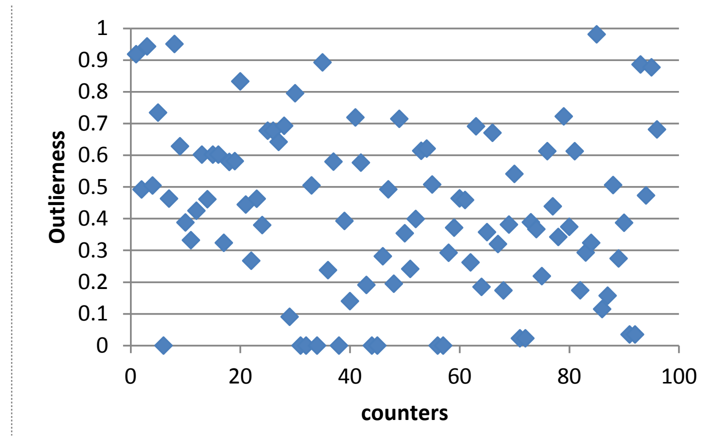


Figure 6.5: Distribution of outlierness/anomaly using MIC

6.2.3 Change Point Detection

The *Change Point Detection* (CPD) [92, 130] is an anomaly detection method where a *"drastic change"* observed from a sequence distribution is detected. These points of sudden change may be indicators of impending possible anomalies. It can be performed as a *real-time* change-point detection or *retrospective* change-point detection [92], depending on suitability. The former deals with detecting real-time changes in applications. A good example is responses in robots. On the other hand, the latter is mostly employed for applications with longer response time and deals with retrospective data.

In this work, we employ the retrospective CPD to detect sudden changes in the utilisation of resources by a supercomputer system. Such sudden changes may point to possible abnormal behaviours in the system. For example, a sudden peak in usage of memory or network resources could signal the presence of errors in the system. Resource usage data are collected and logged as streams of time series data which are formed by the probability distributions of the resources used on a node by running jobs. Therefore, the level at which resources are utilised may vary with time and these changes can be captured using CPD. This is our motivation for using change point detection as an alternative to job anomalies. Specifically, we use change points as anomaly indicators and, since

job anomaly are good failure predictors, by extension we seek to determine the suitability of change points as good failure predictors. We employed the *cumulative sum* change point detection technique in this work, as explained below.

Cumulative Sum Change-Point Detection: Cumulative Sum (CuSUM) CPD approach [130, 131], is based on the fact that sudden change in parameter corresponds to a change in the expected value of log-likelihood ratio. From the name, it tracks the cumulative sums of the differences between the values and the average. At points where the values are above average, the cumulative sum steadily increases. Therefore this method involve finding the mean and its difference with observation values.

Given $S = x_1, x_2, \ldots, x_n$, we first initialise the cumulative sum, $cS_0 = 0$ and obtain the mean of S, given as $\bar{x}$,

$$cS_{i+1} = cS_i + (x_{i+1} - \bar{x}) \tag{6.2}$$

for all $i = 1...n$.

Any abrupt change points are those points with cS_i values above a threshold th.

Preparing data for CPD algorithm: A line of logged usage data contains all the counters and their usage values captured within a certain time period. We refer to these lines of logged usage data as *resource events* (or simply events), e_i[3]. These events are streams of time series data. for the purpose of our research, we capture these events within a given time window, $t_w = 10$ minutes, which we call a subsequence, x_i. A sequence, $S = x_1, x_2, \ldots, x_n$, is then a stream of subsequences, as illustrated in Figure 5.11 (in chapter 5). A reasonably small tw is chosen to avoid capturing different usage patterns within a subsequence and also big enough for such subsequence to be informative.

Hence, each x_i is a vector of the sum of resource usage for each counter.

[3]We refer to both log entries and resource usage data as events. However, the sources of these events will make the nature of the events clear.

The amount of resources used on all the nodes in a subsequence is summed up for each counter. These values are then scaled to values between 0 and 1 to form a probability distribution. This is because the CPD algorithm accepts the data as a probability distribution (this also becomes easier to handle and explained). Figure 6.6 illustrates the outcome of applying CuSUM on a sequence of resource usage data. At point 5, a sudden increase in change point values can be observed, signalling a potential problem in the system.

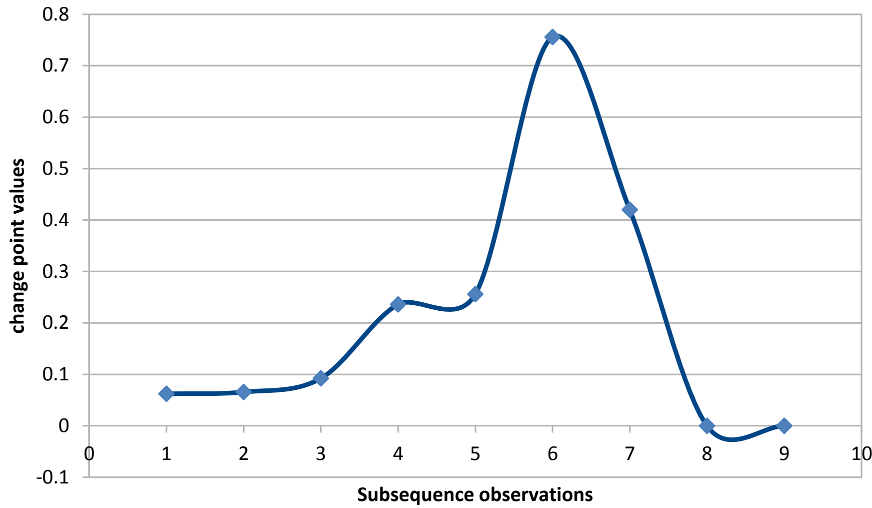


Figure 6.6: Example showing result of CuSUM CPD on a sequence

6.2.4 Lead Times

The aim of any fault analysis is to provide an opportunity for preventive or corrective mechanisms to be taken; that is, system administrators can rectify the problem and/or employ measures that predict or prevent occurrences of failures. The longer the time between the identification of an anomalous behaviour and the time of failure, the higher is the probability of an error handling procedure to complete successfully.

Therefore, from Algorithm 5, we first extract the set $aList$ of anomalous jobs, with the time Tf_{ra} (see Figure 6.7) being the earliest time when an anomalous

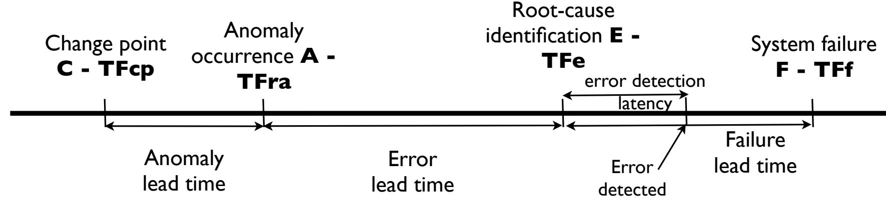


Figure 6.7: Lead Time of Anomalies, Errors and Failures

job has been identified. The time Tf_e (see Figure 6.7) is the first time when an error, diagnosed as a potential root-cause of the system failure F occurring at time Tf_f. The time Tf_{cp} defines the time at which the first change point occurs in the resource usage, i.e., there can be several time periods during which change-points occur and we are only interested in the first such time period.

We obtain the *anomaly, error, failure lead times*, which we denote by $\delta T_c, \delta T_a, \delta T_e$ respectively, as the times between when the earliest change-point (resp. anomalous job and error) is experienced to when an anomaly (resp. a hypothesized root-cause, failure) is first identified, and are defined as follows:

$$\delta T_c = Tf_{ra} - Tf_{cp} \tag{6.3}$$

$$\delta T_a = Tf_e - Tf_{ra} \tag{6.4}$$

$$\delta T_e = Tf_f - Tf_e \tag{6.5}$$

Another important aspect is the relationship between the counters, that may point to a fault, captured by the values of MIC, as explained in Section 6.2.2. The MIC values for a sample of the relationship is shown in Table 6.1. An entry in the table means that, within the time window being considered, for example, two counters, such as *rx_bytes_dropped* and *direct_write* with $MIC \approx 1$, there is high tendency that there is high drop in data when there is an unusually high

data written to storage. Table 6.1 shows the correlation of few pairs of counters with value of $MIC \geq 0.8$.

Table 6.1: Sample Results of Counter correlations (MIC)

counter1	**counter2**	**MIC**
alloc_inode	statfs	0.98425
direct_read	setxattr	1.0
direct_read	getattr	0.96307
dirty_pages_hits	direct_write	0.96166
mmap	alloc_inode	0.93
read_bytes	dirty_pages_hits	1.0
rx_bytes_dropped	direct_write	0.98425
rx_msgs	mmap	0.81772

6.3 Case Study: Ranger Supercomputer

In this section, we demonstrate our proposed methodology on logs and resource usage data obtained from the Ranger supercomputer from the *Texas Advanced Computing Center* (TACC) at the University of Texas at Austin[4] and discuss the results.

6.3.1 Datasets and Performance Measurement

The structure of the datasets have been explained in chapter 3. The data was collected over a period of 4 weeks, with 32GB worth of *resource usage data* and 1.2 GB worth of *rationalized event logs* (or ratlog). The datasets are not labelled, i.e., they are not enhanced with any further failure information than what they already carry.

Performance Measurement: The objective of the paper is to develop a methodology to increase the time window during which error handling procedures can be successfully completed. This increase in time period, which we call the *propagation time*, is given by:

$$propagation\ time = anomaly\ lead\ time + error\ lead\ time \tag{6.6}$$

[4] *www.tacc.utexas.edu*

However, this increase should be linked to situations leading to a possible system failure, i.e., the increased time window should not occur due to a high rate of false positives (good system behaviours wrongly identified as failure pattern).

Thus, to measure the performance of the methodology, we use the so-called *F-measure* metric, which is the harmonic mean of *precision* and *recall*. Precision captures the proportion of patterns that are correctly identified as failure patterns to the total number of patterns identified as failures (correct or not). On the other hand, recall captures the proportion of accurate failure patterns identifications to the total number of actual failure patterns. Specifically, *f-measure* achieves a balance between true and false positives.

6.3.2 Base Case for Comparison - Error Detection Latency using Clustering

We now develop a base case for analyzing the performance of our methodology. We use the methodology developed in chapter 4, which is also seen in [53] for failure detection. The ability to predict failure provides the basis for the development of an error detection mechanism, i.e., once a failure is predicted (with high probability), then a flag indicating the presence of an error can be raised.

We processed the event logs of the Ranger supercomputer as follows: starting from a failure event f, which is typically a *compute node soft lockup* in Ranger, we identified a large enough interval during which there are no overlapping failures, i.e., no two successive failures occur within that time interval. This mean time between failure (MTBF) is at least two hours on Ranger. The reason for this is to prevent the (faulty) behaviour from a previous failure to impact on the behaviour on the following time window.

We then split the log into 2 hour behaviours, some of which end in failures while other are error-free. For each 2-hour behaviour, we then analysed the behaviour in terms of T-min wide slots, i.e., we further split the behaviour into

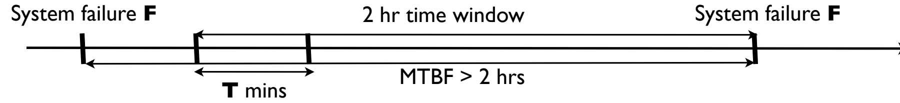


Figure 6.8: Processing and analysis of Ranger event logs.

smaller sequences, so that error detection (hence failure prediction) can be done earlier. This is depicted in Figure 6.8. The results from the detection explain in chapter 4, using the two different distance metrics, are shown in Figure 6.9.

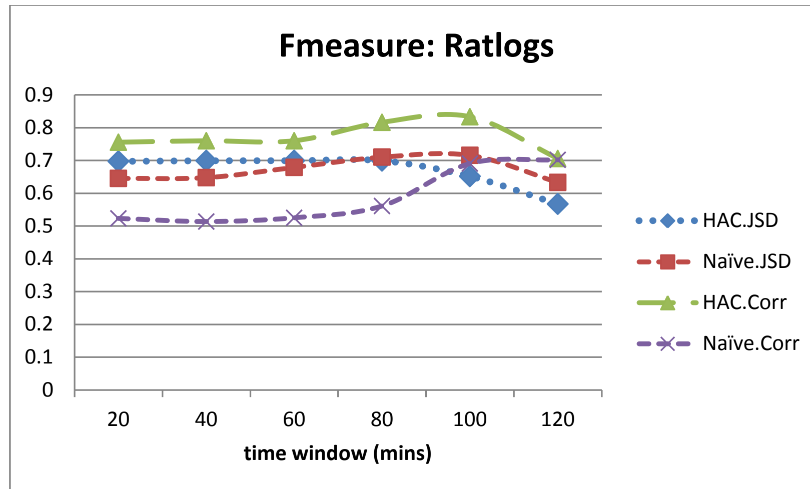


Figure 6.9: Results of clustering algorithms with two different distance metrics.

From Figure 6.9, it can be observed that the hierarchical clustering algorithm achieves better clustering results using the *correlation* distance metric. The f-measure value is consistently higher than 0.7, as opposed to other clustering algorithm and metrics combination.

Error handling time window: We define a failure prediction (hence, error detection) to occur whenever the *f-measure* value exceeds 0.75. This can be observed to occur at approximately 75 minutes within the 2 hour time window,

i.e., the error handling time window is 45 minutes, i.e., $120 - 75 = 45$.

6.3.3 Identifying Anomalies Using our Methodology

From the root-cause analysis of failure on the data, we obtained 872 different fault events from the event logs that are highly correlated with with *soft lock up* failures, which are the frequently occurring failure on Ranger supercomputers. In order to extract the set of anomalous jobs, we define a time window t_w over which resource usage data can be extracted. This time window stretches until the occurrence of the root-cause. For every error event (root-cause), we extract all the resource data within the time interval $[T_e - t_w, T_e]$, forming a resource usage sequence. A small time window means that the "root" anomaly may be missed, while a longer time window may mean that some "trivial" anomaly may be identified that does not lead to failure. The time window t_w used in our work is 60 minutes, as we noticed this was the window with appropriate trade-offs.

Anomalous Jobs Extraction using PCA and ICA

The results of the anomaly detection using PCA and ICA are shown in Table 6.2. As argued previously, we are mostly concerned with the *F-measure* since it captures both precision and recall. As can observed, using PCA and ICA to identify anomalous jobs for failure prediction do not result in the high F-measure (> 0.75) needed. In fact, the F-measure values are around 10% less than the required value. This means that (i) there is a high probability that these anomalous resource usages will not lead to a system failure (ICA) and (ii) any subsequent propagation time is irrelevant due to the possible high proportion of false positives (for PCA). A summary of the anomalous jobs is as shown in Table 6.5.

Table 6.2: Anomaly Detection performance of PCA and ICA.

	Recall	**Precision**	**Fmeasure**
PCA	0.752324	0.61	0.6734
ICA	0.6432	0.73	0.684

Change-Point Detection as Anomaly Detection

Table 6.3: Detection performance for Change Point Detection.

	Recall	**Precision**	**Fmeasure**
CPD	0.67	0.42	0.516

To address the shortcomings in using only anomalies as failure predictors to obtain a significant propagation time, we investigate the use of change points in resource usage as a different type of resource usage anomalies due to sudden surges. The results are shown in Table 6.3. As can be observed, there is a high proportion of false positives (i.e., low precision value). This can be easily explained by the fact that not all change-points will lead to a failure. For example, some change points occur due to a higher workload, within the operational profile of the application. Further, we can conclude that this method cannot be used due to the low F-measure value.

Combining Change-points with PCA or ICA

Table 6.4: Detection performance for Change Point Detection with PCA or ICA.

	Recall	**Precision**	**Fmeasure**
PCA+CPD	0.7914	0.74	0.764
ICA+CPD	0.6932	0.53	0.6023

Given that PCA and ICA give good F-measure values while CPD give worse, we seek to determine whether combining CPD with PCA or ICA gives better result. The intuition behind this is that when a change point occurs followed by an anomaly, this should represent a stronger predictor of failure. As can be observed from Table 6.4, combining CPD and PCA results in a high F-measure value of 0.764. This value is above the failure prediction threshold of 0.75. Thus, we consider that a combination of CPD and PCA will result in a reasonable value for propagation time.

6.3.4 Propagation Time

From the resource usage data, we extract the time at which a resource anomaly is first seen within a sequence. As explained earlier in Section 6.2.4, the time between when an anomalous behaviour is observed to when an error is first logged, δT_a (error lead time), is obtained.

Given that the combination of CPD and PCA gives the best failure prediction with an F-Measure value greater than 0.75 (as for when using the event logs only). The propagation time when using this combination is 3922 seconds (65 minutes). What this means is that, with a sufficiently high probability, these anomalies will lead to failures and that the propagation time is 65 minutes. Hence, from the perspective of error handling, the error handling time window is extended by 65 minutes, which is considerably more than current or predicted future checkpointing time. In comparison to the base case (using clustering), the improvement is approximately $\frac{65}{45} = 140\%$. As an extreme, if an error is detected within the first twenty minutes when using the event logs, then the failure lead time is 100 minutes, giving a worst case improvement of $\frac{65}{100} = 65\%$

For reason of completeness, we provide the propagation time for the different anomaly techniques presented in this paper in Table 6.5.

Table 6.5: Distribution of Anomalous Jobs and Error Propagation Time for all the methods

	aδT_a (seconds)	**No. of Anomalous Jobs Detected**
PCA	3011	16531
ICA	3924	15034
CPD	4022	12064
PCA+CPD	**3922**	**15012**
ICA+CPD	4227	13133

6.3.5 Other Issues

In this section, we investigate two issues concerning the methodology.

Impact of Degree of Outlierness

The first issue is to determine whether there is a correlation between the propagation time (PT) and the anomaly values of a sequence. Specifically, we wish to determine if sequences with high values of "anomaly" have shorter propagation time (hence, a smaller time to failure).

From the graph of Figure 6.10, it can be observed that there is no clear pattern to support the original claim. In other words, the degree of "outlierness" of jobs does not determine the size of the propagation time (hence, the time to failure). This may probably depend of the type of faults and the impact of such faults. This is an area for future work.

Distribution of Propagation Time

The second issue we consider is the distributed of propagation time across different failure sequences. If the distribution of propagation time is such that certain patterns give rise to higher propagation time, then such patterns can be identified to enable better error handling. From Figure 6.11, we observed that for different fault sequences within each week, the propagation time seems to be be within close region, with no much disparity. This might be attributed to the fact that these errors leads to same failure (Soft lockup).

6.4 Summary

In this chapter, we make the following specific contributions:

1. We use two *anomaly-based detection* techniques, namely *principal component analysis* (PCA) and *independent component analysis* (ICA) to identify anomalous resource usage in the system. We label the time when the first resource usage anomaly occurs by Tf_{ra}.

2. We develop an approach based on *change point detection* to observe sudden changes in resource usage, *leading* to a resource usage anomaly. We label

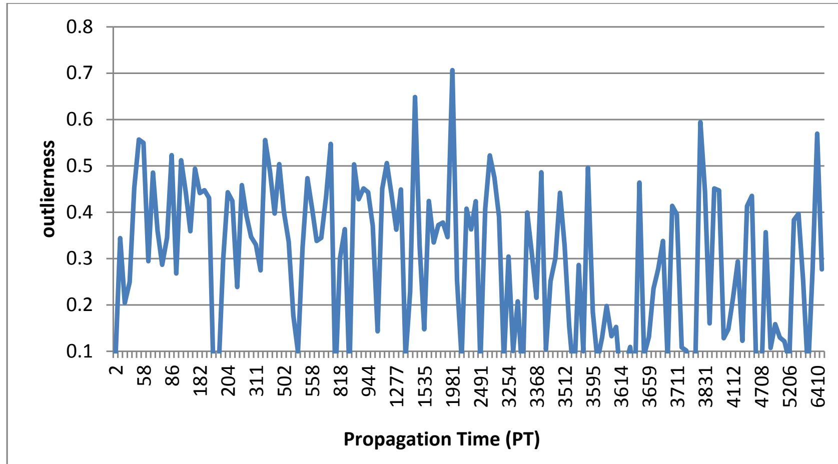


Figure 6.10: Distribution of outlierness/anomaly of different fault sequence for week 1

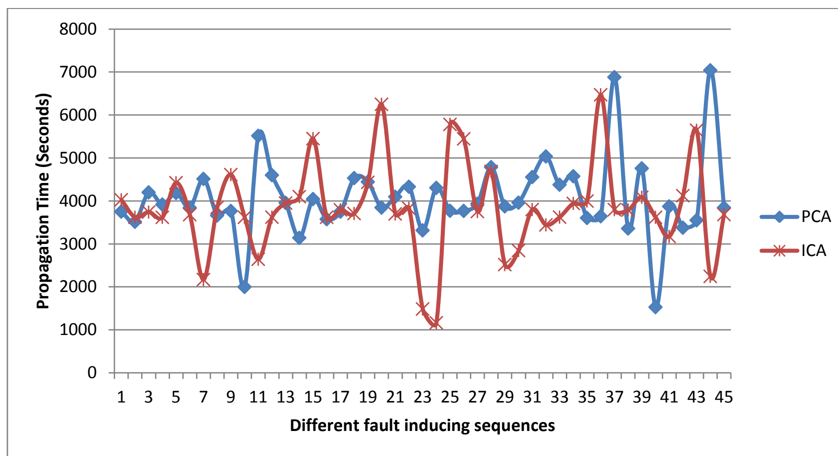


Figure 6.11: Distribution of propagation time for different fault sequences

the time the first sudden burst in resource usage to occur by Tf_{sb} with $Tf_{sb} \leq Tf_{ra}$.

3. Starting with a failure occurring at time T_f, we perform a root-cause analysis of the failure in the system to obtain the time-stamps of the error messages in the logs. We label the time of the first occurrence of an error in the log by Tf_{em} with $Tf_{em} < T_f \wedge Tf_{ra} < Tf_{em}$.

4. We apply our methodology to the resource usage logs and error logs from the Ranger Supercomputer from TACC. Our results show that, in comparison with lead failure times computed when predictions stabilized in [53], the time window can extended by approximately 140%, with a worst-case improvement of around 65%.

CHAPTER 7

Summary, Conclusion and Future Work

7.1 Summary

This thesis is motivated by the challenge of reducing the number of recurrent and costly failures in distributed systems. This necessitated the use of the system's error logs and the resource usage/utilization data for error detection and for enhancing system recovery. In this chapter, we summarize the key points of each chapter, highlighting their key contributions in this thesis with further discussions.

7.1.1 Introductory chapters

These chapters include the Introduction (Chapter 1), the survey of previous methods (Chapter 2) and the description of the system used, the data and fault models (Chapter 3).

The ability to detect the presence of errors that could lead to failure is an important step in fault tolerance. It gives provision for corrective and preventive measures to be taken and in doing this, the thesis utilises three different log data from actual production systems.

The system data is usually huge and can be overwhelming for manual analysis. These huge logs contain events can point to the presence of faults in the system, hence it is wise that we use this data for error detection.

We introduced a taxonomy for fault tolerance methods or approaches. This taxonomy brings to light the categories of the approaches previously published and our approach. Most of the previous work make use of system's event logs to perform detection; however, none was able to use *resource usage data* combined

with event logs for detection as we did in our approach.

We have also learned that not all errors end in failure and detecting these errors can be done in an unsupervised way. Furthermore, different fault categories exist which may have different patterns.

7.1.2 Error Logs Preprocessing and Pattern Detection

As large-scale distributed systems grow in size, the size of the logs which contain information about the system's activities also increases. The huge log size becomes a challenge since logs are the primary source of information for system administrators. However, most of the log events are redundant, i.e., they are not essential for performing failure analysis, therefore the log file can be compressed. We proposed a novel, generic compression algorithm that can be instantiated according to the structure of the log files. The approach filters these logs such that events that serve as precursor to failure are preserved. The method did not only compressed logs, it first extract message types in logs. The clusters formed and indexed with ids represents message types. These message types are useful in log analysis e.g., visualization, indexing etc [95]. Our compression method makes use of event similarity (Levenshtein distance) and event structure to determine a redundant event.

The efficiency of the compression technique is validated through a proposed pattern detection algorithm.

The chapter also addressed the problem of detecting failure inducing error patterns among message logs of large-scale computer systems. We proposed a novel *unsupervised* approach that leverages the characteristics of log patterns and the recurrent interaction between events to detect failure runs. We capture changes in the behaviour of the sequences through change in their entropies. The technique was verified on three different log types with positive results. First, the results demonstrate that compression does not only reduce log size which leads to low computational cost of failure analysis, but also enhances detection of failure patterns. Secondly, runs that end in failure are detected

with an average *F-measure* of 78%.

Future work: As a future work, we intend to develop this approach as *online detection*. We hope to investigate further why this approach is affected by temporal behaviour of faults.

7.1.3 Failure Sequence Detection Using Resource Usage Data and Event Logs

In Chapter 5, we proposed an approach for error detection in large-scale distributed systems. The approach makes use of the novel combination of event logs and resource usage data to improve detection accuracy. Our methodology is based on the computation of (i) mutual information, (ii) entropy and (iii) anomaly score of resource use to determine whether an event sequence is likely to lead to failure, i.e., is erroneous or not. We evaluated our methodology on the logs and resource usage data from the Ranger supercomputer and results are shown to detect errors with a very high accuracy. We compared our approach with Nodeinfo, a well-known error detection methodology, our method outperform Nodeinfo by up to 100%.

In the chapter, we discussed the fact that not all errors lead to failure, i.e., such patterns eventually recover, we call them *recovery patterns*. We realised that most times these patterns are detected as failure inducing (increasing the *false positives*), however, they are not. In our bid to identify such patterns, we proposed a recovery pattern detection method based on change point detection in large-scale distributed systems. The approach makes use of *resource usage data* to detect the recovery sequence among other failure sequences. The method leverages the fact that unusual use of resources by the systems could point to impending failure, to detect recovery patterns. Change point detection is employed to determine the points of anomaly within a sequence. These points of anomalous behaviour points to a recovery or failure sequence. We proposed a detection algorithm based on these parameters to determine if a fault will eventually recover or end in failure. We evaluated our methodology on the

resource usage data from the Ranger supercomputer and the results have shown to detect recovery sequences with good accuracy. It achieved an *F-measure* of 64%.

7.1.4 Increasing the Error Handling Time Window in Large-Scale Distributed Systems

In this Chapter, we have addressed the problem of increasing the error handling or propagation time window. We have thus presented an anomaly detection-based approach to identify anomalous use of resources in a cluster system that points to potentially impending system failure. We have investigated three different anomaly detection methods, namely (i) principal component analysis, (ii) independent component analysis and (iii) change point detection. The method also performs root-cause analysis of failures to identify the time at which the root-cause occurred. We then developed a case study using event logs and resource usage data from the Ranger supercomputer. We implemented the approach from [53] for error detection to obtain a basis for comparison. We found that the combination of CPD and PCA resulted in a high value of F-measure of at least *0.75*, resulting in an extension of the *error handling time window* of 140%, with a worst-case extension of 60%.

7.2 Conclusions

In this thesis, we have proposed an unsupervised error detection approach that leverages on the features inherent in the events that describe faulty and non-faulty patterns in log data. We further developed an approach that extends the error handling time for which error handling techniques can be completed before failure occurs. The huge log data produced by large-scale distributed systems poses great challenge for any automatic analysis. A filtering approach that reduces the size of the data by purging redundant events without losing important ones was proposed. These approaches were applied on data from

production systems and a high detection accuracy was achieved. In comparison with other state-of-the-art methods, ours perform considerably better.

The proposed unsupervised approach comes with some disadvantages: Even though a good detection accuracy was achieved, this accuracy can only be achieved if faults always produce events symptomatic of failure; that is, faults leading to failure must produce events in order for the method to be effective since it is event based. In a case where a fault silently ends in failure, or behaves similar to normal runs, then the performance of the method will be affected negatively. However, the method has demonstrated its ability to perform well across the systems used.

7.3 Future Work

The research work has shown that there is room for improvement and areas where potential research directions could be pursued. We highlight these areas in this chapter.

7.3.1 Improving the Error Detection

Our work demonstrates the usefulness of combining resource usage data with event logs to perform detection. It has also opened up new directions for future research. As a potential future work, the approach can be adapted and implemented as an online detection approach. The implementation can be straightforward, as the event logs and resource usage data can be collected as sequences within a certain time window as they are being logged. This can be implemented on a live production system for testing.

As an improvement, a further investigation into the performance of this approach on different system data and different time windows can be done; especially, more resource usage data from other systems can be used.

We believe that the features extracted from sequences can be used to perform a supervised learning method for error prediction. This is a promising research

direction where the temporal relationships between events can be captured and incorporated as features. It will enable supervised error prediction using approaches such as: *Hidden Markov Model, Support Vector Machines (SVM)* and *Conditional Random Fields (CRF)* among many others.

7.3.2 Improving the Recovery Run Detection

One challenge with this method is that few data is available for testing. The experiment was done on data from one production system with few recovery patterns. A more firm assertion can be made about this technique if more data is used from different production systems. As as future work, the behaviours of nodes and the jobs running on each node can be studied for correlation. This can give an insight to their behaviour and possible recovery.

7.3.3 The Error Handling Time

One of the key findings in this work is that the best way to improve the time window for which error handling techniques like checkpointing or task migration can be effectively applied is to be able to trigger these techniques early enough when failure symptoms are noticed. Therefore, other methods for detecting early fault symptoms can be implemented so that preventive error handling methods can be applied.

Bibliography

[1] 1003.1 standard for information technology portable operating system interface (posix) rationale (informative). *IEEE Std 1003.1-2001. Rationale (Informative)*, pages i–310, 2001. doi: 10.1109/IEEESTD.2001.93367.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8.

[3] M. Aharon, G. Barash, I. Cohen, and E. Mordechai. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part I*, ECML PKDD '09, pages 227–243, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04179-2.

[4] A. Avižienis. Design of fault-tolerant computers. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 733–743, New York, NY, USA, 1967. ACM. doi: 10.1145/1465611.1465708.

[5] A. Avizienis and J.-C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986. ISSN 0018-9219. doi: 10.1109/PROC.1986.13527.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.

[7] C. Bai, Q. Hu, M. Xie, and S. Ng. Software failure prediction based on a markov bayesian network model. *Journal of Systems and Software*, 74

(3):275–282, Feb. 2005. ISSN 0164-1212. doi: 10.1016/j.jss.2004.02.028. URL http://dx.doi.org/10.1016/j.jss.2004.02.028.

[8] M. Barborak, A. Dahbura, and M. Malek. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, 25(2):171–220, June 1993. ISSN 0360-0300. doi: 10.1145/152610.152612. URL http://doi.acm.org/10.1145/152610.152612.

[9] E. Berrocal, L. Yu, S. Wallace, M. Papka, and Z. Lan. Exploring void search for fault detection on extreme scale systems. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 1–9, Sept 2014. doi: 10.1109/CLUSTER.2014.6968757.

[10] M. W. Berry, Z. Drmac, Elizabeth, and R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41:335–362, 1999.

[11] M. Bouguerra, A. Gainaru, L. Gomez, F. Cappello, S. Matsuoka, and N. Maruyama. Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 501–512, May 2013. doi: 10.1109/IPDPS.2013.74.

[12] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 93–104, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4. doi: 10.1145/342009.335388. URL http://doi.acm.org/10.1145/342009.335388.

[13] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, Dec. 1992. ISSN 0891-2017. URL http://dl.acm.org/citation.cfm?id=176313.176316.

[14] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *IJHPCA*, 23(4):374–388, 2009.

[15] F. Cappello, H. Casanova, and Y. Robert. Checkpointing vs. migration for post-petascale supercomputers. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 168–177, Sept 2010. doi: 10.1109/ICPP.2010.26.

[16] F. Cappello, H. Casanova, and Y. Robert. Preventive migration vs. preventive checkpointing for extreme scale supercomputers. *Parallel Processing Letters*, 21(2):111–132, 2011.

[17] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 24(5):823–839, May 2012. ISSN 1041-4347. doi: 10.1109/TKDE.2010.235.

[18] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1): 63–75, Feb. 1985. ISSN 0734-2071. doi: 10.1145/214451.214456. URL `http://doi.acm.org/10.1145/214451.214456`.

[19] H. Chen, G. Jiang, and K. Yoshihira. Failure detection in large-scale internet services by principal subspace mapping. *Knowledge and Data Engineering, IEEE Transactions on*, 19(10):1308–1320, Oct 2007. ISSN 1041-4347. doi: 10.1109/TKDE.2007.190633.

[20] X. Chen, C.-D. Lu, and K. Pattabiraman. Predicting job completion times using system logs in supercomputing clusters. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–8, June 2013. doi: 10.1109/DSNW.2013.6615513.

[21] Z. Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 73–84, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0552-5. doi: 10.1145/1996130.1996142.

[22] J.-F. Chiu and G. ming Chiu. Placing forced checkpoints in distributed real-time embedded systems. *Computing & Control Engineering Journal*, 13:200–207, 2002.

[23] E. Chuah, S. hao Kuo, P. Hiew, W.-C. Tjhi, G. Lee, J. Hammond, M. Michalewicz, T. Hung, and J. Browne. Diagnosing the root-causes of failures from cluster log files. In *2010 International Conference High Performance Computing (HiPC)*, pages 1 –10, dec. 2010.

[24] E. Chuah, G. Lee, W.-C. Tjhi, S.-H. Kuo, T. Hung, J. Hammond, T. Minyard, and J. C. Browne. Establishing hypothesis for recurrent system failures from cluster log files. In *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, DASC '11, pages 15–22, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4612-4.

[25] E. Chuah, A. Jhumka, S. Narasimhamurthy, J. Hammond, J. C. Browne, and B. Barth. Linking resource usage anomalies with system failures from cluster log data. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 111–120, 2013. doi: 10.1109/SRDS.2013.20.

[26] G. Cormode. The continuous distributed monitoring model. *SIGMOD Rec.*, 42(1):5–14, May 2013. ISSN 0163-5808. doi: 10.1145/2481528.2481530. URL http://doi.acm.org/10.1145/2481528.2481530.

[27] K. Das, J. Schneider, and D. B. Neill. Anomaly pattern detection in categorical datasets. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 169–176, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-193-4. doi: 10.1145/1401890.1401915. URL http://doi.acm.org/10.1145/1401890.1401915.

[28] S. Das and P. Suganthan. Differential evolution: A survey of the state-of-the-art. *Evolutionary Computation, IEEE Transactions on*, 15(1):4–31, Feb 2011. ISSN 1089-778X. doi: 10.1109/TEVC.2010.2059031.

[29] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 162–171, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0102-2. doi: 10.1145/1995896.1995923.

[30] W. J. Dixon and A. M. Mood. The statistical sign test. *Journal of the American Statistical Association*, Vol. 41(236):557– 566, Dec 1946.

[31] S. T. Dumais, G. Furnas, T. Landauer, S. Deerwester, et al. Latent semantic indexing. In *Proceedings of the Text Retrieval Conference*, 1995.

[32] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *Journal of Supercomputing*, 65(3):1302–1326, Sept. 2013. ISSN 0920-8542. doi: 10.1007/s11227-013-0884-0. URL `http://dx.doi.org/10.1007/s11227-013-0884-0`.

[33] N. El-Sayed and B. Schroeder. To checkpoint or not to checkpoint: Understanding energy-performance-i/o tradeoffs in HPC checkpointing. In *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*, pages 93–102, 2014. doi: 10.1109/CLUSTER.2014.6968778.

[34] D. Endres and J. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, July 2003. ISSN 0018-9448. doi: 10.1109/TIT.2003.813506.

[35] T. Fawcett and F. Provost. Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD

'99, pages 53–62, New York, NY, USA, 1999. ACM. ISBN 1-58113-143-7. doi: 10.1145/312129.312195. URL http://doi.acm.org/10.1145/312129.312195.

[36] I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko. Failure prediction based on log files using random indexing and support vector machines. *J. Syst. Softw.*, 86(1):2–11, Jan. 2013. ISSN 0164-1212. doi: 10.1016/j.jss.2012.06.025. URL http://dx.doi.org/10.1016/j.jss.2012.06.025.

[37] S. Fu and C.-Z. Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12, Nov 2007. doi: 10.1145/1362622.1362678.

[38] X. Fu, R. Ren, J. Zhan, W. Zhou, Z. Jia, and G. Lu. Logmaster: Mining event correlations in logs of large-scale cluster systems. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 71–80, Oct 2012. doi: 10.1109/SRDS.2012.40.

[39] E. W. Fulp, G. A. Fink, and J. N. Haack. Predicting computer system failures using support vector machines. In *Proceedings of the First USENIX Conference on Analysis of System Logs*, WASL'08, pages 5–5, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855886.1855891.

[40] M. Gabel, A. Schuster, R.-G. Bachrach, and N. Bjorner. Latent fault detection in large scale services. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1624-8. URL http://dl.acm.org/citation.cfm?id=2354410.2355160.

[41] A. Gainaru, F. Cappello, J. Fullop, S. Trausan-Matu, and W. Kramer. Adaptive event prediction strategy with dynamic time window for large-

scale hpc systems. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML '11, pages 4:1–4:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0978-3.

[42] A. Gainaru, F. Cappello, S. Trausan-Matu, and B. Kramer. Event log mining tool for large scale hpc systems. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 52–64, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23399-9. URL http://dl.acm.org/citation.cfm?id=2033345.2033352.

[43] A. Gainaru, F. Cappello, and W. Kramer. Taming of the shrew: Modeling the normal and faulty behaviour of large-scale hpc systems. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1168–1179, May 2012. doi: 10.1109/IPDPS.2012.107.

[44] A. Gainaru, F. Cappello, M. Snir, and W. Kramer. Fault prediction under the microscope: A closer look into hpc systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 77:1–77:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5.

[45] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. Scheduling the I/O of HPC applications under congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 1013–1022, 2015. doi: 10.1109/IPDPS.2015.116. URL http://dx.doi.org/10.1109/IPDPS.2015.116.

[46] P. Garraghan, P. Townend, and J. Xu. An empirical failure-analysis of a large-scale cloud computing environment. In *High-Assurance Systems*

Engineering (HASE), 2014 IEEE 15th International Symposium on, pages 113–120, Jan 2014. doi: 10.1109/HASE.2014.24.

[47] J. J. Gertler. Survey of model-based failure detection and isolation in complex plants. *Control Systems Magazine, IEEE*, 8(6):3–11, Dec 1988. ISSN 0272-1708. doi: 10.1109/37.9163.

[48] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601902.

[49] J. Gu, Z. Zheng, Z. Lan, J. White, E. Hocks, and B.-H. Park. Dynamic meta-learning for failure prediction in large-scale systems: A case study. In *37th International Conference on Parallel Processing, ICPP '08.*, pages 157–164, 2008.

[50] Q. Guan, Z. Zhang, and S. Fu. Proactive failure management by integrated unsupervised and semi-supervised learning for dependable cloud systems. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 83–90, 2011.

[51] T.-H. Guo and J. Nurre. Sensor failure detection and recovery by neural networks. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume i, pages 221–226 vol.1, Jul 1991. doi: 10.1109/IJCNN.1991.155180.

[52] A. Gupta, B. Acun, O. Sarood, and L. Kale. Towards realizing the potential of malleable jobs. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10, Dec 2014. doi: 10.1109/HiPC.2014.7116905.

[53] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne. Towards detecting patterns in failure logs of large-scale distributed systems. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2015 IEEE International*. IEEE, 2015.

[54] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne. Towards increasing the error handling time window in large-scale distributed systems using console and resource usage logs. In *Proceedings of The 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA 2015)*, Aug 2015.

[55] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne. On the impact of redundancy handling in event logs on classification in cluster systems. In *Proceedings of International Conference on Dependability (DEPEND)*, Aug 2015.

[56] D. Hakkarinen and Z. Chen. Multilevel diskless checkpointing. *Computers, IEEE Transactions on*, 62(4):772–783, April 2013. ISSN 0018-9340. doi: 10.1109/TC.2012.17.

[57] D. Hakkarinen, P. Wu, and Z. Chen. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *Parallel and Distributed Systems, IEEE Transactions on*, 26(5):1323–1335, May 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2014.2320502.

[58] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. Clustering validity checking methods: Part ii. *SIGMOD Rec.*, 31(3):19–27, Sept. 2002. ISSN 0163-5808. doi: 10.1145/601858.601862. URL `http://doi.acm.org/10.1145/601858.601862`.

[59] J. Hammond. Tacc_stats: I/o performance monitoring for the intransigent. In *In Invited Keynote for the 3rd IASDS Workshop*, 2011.

[60] J. L. Hammond, T. Minyard, and J. Browne. End-to-end framework for fault management for open source clusters: Ranger. In *Proceedings of the 2010 TeraGrid Conference*, TG '10, pages 9:1–9:6, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-818-6.

[61] J. Hansen and D. Siewiorek. Models for time coalescence in event logs.

In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 221 –227, jul 1992.

[62] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 45:1–45:11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063444. URL `http://doi.acm.org/10.1145/2063384.2063444`.

[63] J. Hong, C.-C. Liu, and M. Govindarasu. Integrated anomaly detection for cyber security of the substations. *Smart Grid, IEEE Transactions on*, 5(4):1643–1653, July 2014. ISSN 1949-3053. doi: 10.1109/TSG.2013.2294473.

[64] S. Hussain, M. Mokhtar, and J. Howe. Sensor failure detection, identification, and accommodation using fully connected cascade neural network. *Industrial Electronics, IEEE Transactions on*, 62(3):1683–1692, March 2015. ISSN 0278-0046. doi: 10.1109/TIE.2014.2361600.

[65] A. Hyvärinen. Independent component analysis: recent advances. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 371(1984), 2012. ISSN 1364-503X. doi: 10.1098/rsta.2011.0534.

[66] A. Hyvärinen and E. Oja. Independent component analysis: Algorithms and applications. *Neural Netw.*, 13(4-5):411–430, May 2000. ISSN 0893-6080. doi: 10.1016/S0893-6080(00)00026-5. URL `http://dx.doi.org/10.1016/S0893-6080(00)00026-5`.

[67] E. M. III, S. Speakman, and D. B. Neill. Fast generalized subset scan for anomalous pattern detection. *Journal of Machine Learning Research*, 14:

1533–1561, 2013. URL `http://jmlr.org/papers/v14/mcfowland13a.html`.

[68] R. Iyer, L. Young, and P. Rosetti. Automatic recognition of intermittent failures: an experimental study of field data. *Computers, IEEE Transactions on*, 39(4):525 –537, apr 1990. ISSN 0018-9340.

[69] R. K. Iyer, L. T. Young, and V. Sridhar. Recognition of error symptoms in large systems. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 797–806, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press. ISBN 0-8186-4743-4.

[70] S. Jain, I. Singh, A. Chandra, Z.-L. Zhang, and G. Bronevetsky. Extracting the textual and temporal structure of supercomputing logs. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 254–263, Dec 2009. doi: 10.1109/HIPC.2009.5433202.

[71] I. Jangjaimon and N.-F. Tzeng. Adaptive incremental checkpointing via delta compression for networked multicore systems. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 7–18, May 2013. doi: 10.1109/IPDPS.2013.33.

[72] J. Jiang and L. Yasakethu. Anomaly detection via one class svm for protection of scada systems. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2013 International Conference on*, pages 82–88, Oct 2013. doi: 10.1109/CyberC.2013.22.

[73] S. Kalaiselvi and V. Rajaraman. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana*, 25(5):489–510, 2000. ISSN 0256-2499. doi: 10.1007/BF02703630. URL `http://dx.doi.org/10.1007/BF02703630`.

[74] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Con-*

ference on File and Storage Technologies, FAST'10, pages 4–14, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855511.1855515`.

[75] J. Kittler, W. Christmas, T. de Campos, D. Windridge, F. Yan, J. Illingworth, and M. Osman. Domain anomaly detection in machine perception: A system architecture and taxonomy. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(5):845–859, May 2014. ISSN 0162-8828. doi: 10.1109/TPAMI.2013.209.

[76] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, and Others. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, University of Notre Dame, CSE Dept., 2008.

[77] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951. doi: 10.1214/aoms/1177729694. URL `http://dx.doi.org/10.1214/aoms/1177729694`.

[78] K. P. K. Kumar and R. C. Hansdah. An efficient and scalable checkpointing and recovery algorithm for distributed systems. In S. Chaudhuri, S. R. Das, H. S. Paul, and S. Tirthapura, editors, *ICDCN*, volume 4308 of *Lecture Notes in Computer Science*, pages 94–99. Springer, 2006. ISBN 3-540-68139-6.

[79] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. *SIGCOMM Computer Communication Review*, 35 (4):217–228, Aug. 2005. ISSN 0146-4833. doi: 10.1145/1090191.1080118. URL `http://doi.acm.org/10.1145/1090191.1080118`.

[80] Z. Lan, Z. Zheng, and Y. Li. Toward automated anomaly identification in large-scale systems. *Parallel and Distributed Systems, IEEE Transactions on*, 21(2):174 –187, feb. 2010. ISSN 1045-9219.

[81] J.-C. Laprie. Dependable computing: Concepts, challenges, directions. In *COMPSAC*, page 242, 2004.

[82] A. Lavinia, C. Dobre, F. Pop, and V. Cristea. A failure detection system for large scale distributed systems. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 482–489, Feb 2010. doi: 10.1109/CISIS.2010.29.

[83] C. Lee, R. Alena, and P. Robinson. Migrating fault trees to decision trees for real time fault detection on international space station. In *Aerospace Conference, 2005 IEEE*, pages 1–6, March 2005. doi: 10.1109/AERO.2005.1559584.

[84] Y.-J. Lee, Y.-R. Yeh, and Y.-C. F. Wang. Anomaly detection via online oversampling principal component analysis. *Knowledge and Data Engineering, IEEE Transactions on*, 25(7):1460–1470, July 2013. ISSN 1041-4347. doi: 10.1109/TKDE.2012.99.

[85] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[86] K. Li, J. Naughton, and J. Planck. Checkpointing multicomputer applications. In *Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on*, pages 2–11, Sep 1991. doi: 10.1109/RELDIS.1991.145398.

[87] P. Liang. Semi-supervised learning for natural language. Master's thesis, MIT, 2005.

[88] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. Sahoo, J. Moreira, and M. Gupta. Filtering failure logs for a bluegene/l prototype. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 476–485, June 2005. doi: 10.1109/DSN.2005.50.

[89] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *Dependable Systems and*

Networks, 2006. DSN 2006. International Conference on, pages 425–434, June 2006. doi: 10.1109/DSN.2006.18.

[90] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, DSN 2008.*, pages 398–403, 2008.

[91] J. Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991.

[92] S. Liu, M. Yamada, N. Collier, and M. Sugiyama. Change-point detection in time-series data by relative density-ratio estimation. *Neural Netw.*, 43: 72–83, July 2013. ISSN 0893-6080. doi: 10.1016/j.neunet.2013.01.012. URL `http://dx.doi.org/10.1016/j.neunet.2013.01.012`.

[93] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. An evaluation of entropy based approaches to alert detection in high performance cluster logs. In *Proceedings of the 7th International Conference on Quantitative Evaluation of SysTems(QEST)*. IEEE, 2010.

[94] A. Makanju, A. Zincir-Heywood, and E. Milios. System state discovery via information content clustering of system logs. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 301–306, Aug 2011. doi: 10.1109/ARES.2011.51.

[95] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. A lightweight algorithm for message type extraction in system application logs. *IEEE Trans. on Knowl. and Data Eng.*, 24(11):1921–1936, Nov. 2012. ISSN 1041-4347. doi: 10.1109/TKDE.2011.138. URL `http://dx.doi.org/10.1109/TKDE.2011.138`.

[96] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Fast entropy based alert detection in supercomputer logs. In *PFARM '10: Proceedings*

of the 2nd DSN Workshop on Proactive Failure Avoidance, Recovery and Maintenance (PFARM). IEEE, 2010.

[97] A. Mehri, M. Jamaati, and H. Mehri. Word ranking in a single document by jensenshannon divergence. *Physics Letters A*, 379(2829):1627 – 1632, 2015. ISSN 0375-9601. doi: http://dx.doi.org/10.1016/j.physleta.2015.04.030. URL `http://www.sciencedirect.com/science/article/pii/S0375960115003722`.

[98] P. Munk, B. Saballus, J. Richling, and H.-U. Heiss. Position paper: Real-time task migration on many-core processors. In *Architecture of Computing Systems. Proceedings, ARCS 2015 - The 28th International Conference on*, pages 1–4, March 2015.

[99] Y. Murphey, M. Masrur, Z. Chen, and B. Zhang. Model-based fault diagnosis in electric drives using machine learning. *Mechatronics, IEEE/ASME Transactions on*, 11(3):290–303, June 2006. ISSN 1083-4435. doi: 10.1109/TMECH.2006.875568.

[100] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado. Machine learning methods for predicting failures in hard drives: A multiple-instance application. *J. Mach. Learn. Res.*, 6:783–816, Dec. 2005. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1046920.1088699`.

[101] S. W. Neville, B. Eng, and M. A. Sc. Approaches for early fault detection in large scale engineering plants, 1998.

[102] A. Oliner and A. Aiken. Online detection of multi-component interactions in production systems. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 49–60, June 2011. doi: 10.1109/DSN.2011.5958206.

[103] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *International Conference on Dependable Systems and Net-*

works, 2007. DSN '07. 37th Annual IEEE/IFIP, pages 575 –584, june 2007.

[104] A. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, Feb. 2012. ISSN 0001-0782. doi: 10.1145/2076450.2076466.

[105] A. J. Oliner and A. Aiken. A query language for understanding component interactions in production systems. In *Proceedings of the 24th International Conference on Supercomputing, 2010, Tsukuba, Ibaraki, Japan, June 2-4, 2010*, pages 201–210, 2010. doi: 10.1145/1810085.1810114. URL http://doi.acm.org/10.1145/1810085.1810114.

[106] A. J. Oliner, A. Aiken, and J. Stearley. Alert detection in system logs. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 959–964. IEEE, 2008.

[107] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, pages 191–200, 2010. doi: 10.1109/DSN.2010.5544921. URL http://doi.ieeecomputersociety.org/10.1109/DSN.2010.5544921.

[108] X. Ouyang, S. Marcarelli, R. Rajachandrasekar, and D. Panda. Rdma-based job migration framework for mpi over infiniband. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 116–125, Sept 2010. doi: 10.1109/CLUSTER.2010.20.

[109] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. Iyer. Improving log-based field failure data analysis of multi-node computing systems. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 97–108, June 2011. doi: 10.1109/DSN.2011.5958210.

[110] S. Perarnau, R. Thakur, K. Iskra, K. Raffenetti, F. Cappello, R. Gupta, P. H. Beckman, M. Snir, H. Hoffmann, M. Schulz, and B. Rountree. Distributed monitoring and management of exascale systems in the argo project. In *Distributed Applications and Interoperable Systems - 15th IFIP WG 6.1 International Conference, DAIS 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, pages 173–178, 2015. doi: 10.1007/978-3-319-19129-4_14.

[111] R. Rajachandrasekar, X. Besseron, and D. Panda. Monitoring and predicting hardware failures in hpc clusters with ftb-ipmi. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1136–1143, May 2012. doi: 10.1109/IPDPSW.2012.139.

[112] X. Rao, H. Wang, D. Shi, Z. Chen, H. Cai, Q. Zhou, and T. Sun. Identifying faults in large-scale distributed systems by filtering noisy error logs. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 140–145, June 2011. doi: 10.1109/DSNW.2011.5958800.

[113] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti. Detecting novel associations in large data sets. *Science*, 334(6062):1518–1524, 2011. doi: 10.1126/science.1205438.

[114] S. Sabato, E. Yom-Tov, A. Tsherniak, and S. Rosset. Analyzing system logs: A new view of what is important. In *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, pages 1–7. USENIX Association, 2007.

[115] F. Salfner. *Event-based Failure Prediction: An Extended Hidden Marrkov*

Model Approach. PhD thesis, Humboldt Universitat zu Berlin, Germany, 2008.

[116] F. Salfner and S. Tschirpke. Error log processing for accurate failure prediction. In *in 1st UNIX Workshop on the Analysis of System Logs*, December 2008.

[117] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive logfiles for autonomic systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 211–218, 2004.

[118] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3):10:1–10:42, Mar. 2010. ISSN 0360-0300.

[119] T. Sandhan, T. Srivastava, A. Sethi, and J. Y. Choi. Unsupervised learning approach for abnormal event detection in surveillance video by revealing infrequent patterns. In *Image and Vision Computing New Zealand (IVCNZ), 2013 28th International Conference of*, pages 494–499, Nov 2013. doi: 10.1109/IVCNZ.2013.6727064.

[120] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2607-1. doi: 10.1109/DSN.2006.5. URL `http://dx.doi.org/10.1109/DSN.2006.5`.

[121] R. Selmic and F. Lewis. Multimodel neural networks identification and failure detection of nonlinear systems. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 4, pages 3128–3133 vol.4, 2001. doi: 10.1109/.2001.980299.

[122] C. Shannon. A mathematical theory of communication. *Bell System*

Technical Journal, The, 27(3):379–423, July 1948. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[123] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems (3rd Ed.): Design and Evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998. ISBN 1-56881-092-X.

[124] L. Silva and J. Silva. Using two-level stable storage for efficient checkpointing. *IEEE Software Engineering Journal*, 145(6), 1998.

[125] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *IJHPCA*, 28(2):129–173, 2014. doi: 10.1177/1094342014522573. URL `http://dx.doi.org/10.1177/1094342014522573`.

[126] M. Sonoda, Y. Watanabe, and Y. Matsumoto. Prediction of failure occurrence time based on system log message pattern learning. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 578 –581, april 2012.

[127] C. Spitz and A. Koehler. Tips and tricks for diagnosing lustre problems on cray systems. In *CUG 2011 Proceedings*, 2011.

[128] J. Stearley. Towards informatic analysis of syslogs. In *Cluster Computing, 2004 IEEE International Conference on*, pages 309–318, 2004.

[129] J. Stearley and A. J. Oliner. Bad words: Finding faults in spirit's syslogs. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 765–770. IEEE, 2008.

[130] A. Strong. A review of anomaly detection with focus on changepoint detection. Master's thesis, Department of Mathematics , Swiss Federal Institute of Technology Zurich, 2012.

[131] W. A. Taylor. Change-point analysis: A powerful new tool for detecting changes, 2000.

[132] T.-T. Teoh, S.-Y. Cho, and Y.-Y. Nguwi. Hidden markov model for hard-drive failure detection. In *Computer Science Education (ICCSE), 2012 7th International Conference on*, pages 3–8, July 2012. doi: 10.1109/ICCSE. 2012.6295014.

[133] J. W. Tukey. Mathematics and the Picturing of Data. In R. D. James, editor, *International Congress of Mathematicians 1974*, volume 2, pages 523–532, 1974.

[134] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IP Operations Management, 2003. (IPOM 2003). 3rd IEEE Workshop on*, pages 119–126, 2003. doi: 10.1109/IPOM.2003.1251233.

[135] R. Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *In Proceedings of the 2004 IFIP International Conference on Intelligence in Communication Systems*, pages 293–308, 2004.

[136] N. H. Vaidya. A case for two-level distributed recovery schemes. In *In ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 64–73, 1995.

[137] K. Vaidyanathan and K. S. Trivedi. A measurement-based model for estimation of resource exhaustion in operational software systems. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, ISSRE '99, pages 84–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0443-4. URL `http://dl.acm.org/citation.cfm?id=851020.856189`.

[138] S. Venkatesan. Message-optimal incremental snapshots. In *Distributed Computing Systems, 1989., 9th International Conference on*, pages 53–60, Jun 1989. doi: 10.1109/ICDCS.1989.37930.

[139] R. Vilalta and S. Ma. Predicting rare events in temporal domains. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 474–481, 2002. doi: 10.1109/ICDM.2002.1183991.

[140] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, 34:43–99, 1956.

[141] D. Vounckx, G. Deconinck, J. Vounckx, R. Lauwereins, and J. A. Peperstraete. Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. *International Journal of Modeling and Simulation*, 18:262–265, 1993.

[142] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Proactive process-level live migration in hpc environments. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, Nov 2008. doi: 10.1109/SC.2008.5222634.

[143] A. S. Willsky. Paper: A survey of design methods for failure detection in dynamic systems. *Automatica*, 12(6):601–611, Nov. 1976. ISSN 0005-1098. doi: 10.1016/0005-1098(76)90041-8. URL `http://dx.doi.org/10.1016/0005-1098(76)90041-8`.

[144] Z. Xinmin, Y. Xiaochun, Z. Chen, and S. Jinwei. Artificial neural network for sensor failure detection in an automotive engine. In *Instrumentation and Measurement Technology Conference, 1994. IMTC/94. Conference Proceedings. 10th Anniversary. Advanced Technologies in I amp; M., 1994 IEEE*, pages 167–170 vol.1, May 1994. doi: 10.1109/IMTC.1994.352099.

[145] J. Xu, B. Randell, A. Romanovsky, R. Stroud, A. Zorzo, E. Canver, and F. von Henke. Rigorous development of an embedded fault-tolerant system

based on coordinated atomic actions. *Computers, IEEE Transactions on*, 51(2):164–179, Feb 2002. ISSN 0018-9340. doi: 10.1109/12.980006.

[146] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Mining console logs for large-scale system problem detection. In *Workshop on Tackling Computer Problems with Machine Learning Techniques (SysML), San Diego, CA*, 2008.

[147] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3.

[148] K. Yamanishi and Y. Maruyama. Dynamic syslog mining for network failure monitoring. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 499–508, New York, NY, USA, 2005. ACM. ISBN 1-59593-135-X. doi: 10.1145/1081870.1081927. URL `http://doi.acm.org/10.1145/1081870.1081927`.

[149] K. Yamanishi, J. I. Takeuchi, G. Williams, and P. Milne. On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 320–324, New York, NY, USA, 2000. ACM. ISBN 1-58113-233-6. doi: 10.1145/347090.347160.

[150] Y. Yulevich, A. Pyasik, and L. Gorelik. Anomaly detection algorithms on ibm infosphere streams: Anomaly detection for data in motion. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 301–308, July 2012. doi: 10.1109/ISPA.2012.145.

[151] J. Zhang. Advancements of outlier detection: A survey. *EAI Endorsed Trans. Scalable Information Systems*, 1:e2, 2013. doi: 10.4108/trans.sis.2013.01-03.e2. URL `http://dx.doi.org/10.4108/trans.sis.2013.01-03.e2`.

[152] Y. Zhang and A. Sivasubramaniam. Failure prediction in ibm bluegene/l event logs. In *Parallel and Distributed Processing,. IPDPS 2008. IEEE International Symposium on*, pages 1–5, April 2008. doi: 10.1109/IPDPS.2008.4536397.

[153] Y. Zhao, X. Liu, S. Gan, and W. Zheng. Predicting disk failures with hmm- and hsmm-based approaches. In P. Perner, editor, *Advances in Data Mining. Applications and Theoretical Aspects*, volume 6171 of *Lecture Notes in Computer Science*, pages 390–404. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14399-1. doi: 10.1007/978-3-642-14400-4_30. URL `http://dx.doi.org/10.1007/978-3-642-14400-4_30`.

[154] Z. Zheng, Z. Lan, B. H. Park, and A. Geist. System log pre-processing to improve failure prediction. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 572–577. IEEE, 2009.

[155] B. Zhu, G. Wang, X. Liu, D. Hu, S. Lin, and J. Ma. Proactive drive failure prediction for large scale storage systems. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–5, May 2013. doi: 10.1109/MSST.2013.6558427.