Problem Determination In Message-Flow Internet Services Based On Statistical Analysis of Event Logs

by

Yu Xu

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

Waterloo, Ontario, Canada, 2009

AUTHOR'S DECLARATION

Abstract

In a message-flow Internet service where messages travel through multiple nodes, event log analysis is one of the most important methods to identify the root causes of problems. Traditional approaches for event log analysis have been largely based on expert systems that build static dependency models on rules and patterns defined by human experts. However, the semantic complexity and the various formats of event logs make it difficult to be modeled. In addition, it is time consuming to maintain such static model for constantly evolving Internet services. Recent research has been focused on building statistical models. However, all of these models rely on the trace information provided by J2EE or .NET frameworks, which are not available to all Internet services.

In this thesis, we propose a framework of problem determination based on statistical analysis of event logs. We assume a unique message ID will be logged in multiple log lines to trace the message flow in the system. A generic log adaptor is defined to extract valuable information from the log entries. We also develop an algorithm of log event clustering and log pattern clustering. Frequency analysis will be performed based on the log patterns in order to build a statistical model of the system behaviors. Once the system is modeled, we can determine problems by running a chi-square goodness of fit test using a sliding window approach. As event logs are available on all major operating systems, we believe our framework is a generic solution for problem determination in message-flow Internet services.

Our solution has been validated by the log data collected from the Blackberry Internet Service (BIS) engine [4] , a wireless email service that serves millions of users across the world. According to the test results, our solution shows high accuracy of problem determination.

Acknowledgements

I would like to thank my advisor, Dr. Paul A.S. Ward, for his precious guidance and continuous support throughout this work. I would also thank my family and my managers Andy Van and Kevin McEachern in the BIS engine team for their support and assistance.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Today’s large and dynamic Internet services are usually composed of multiple components. In such systems, messages can pass through different components and create interactions among them.

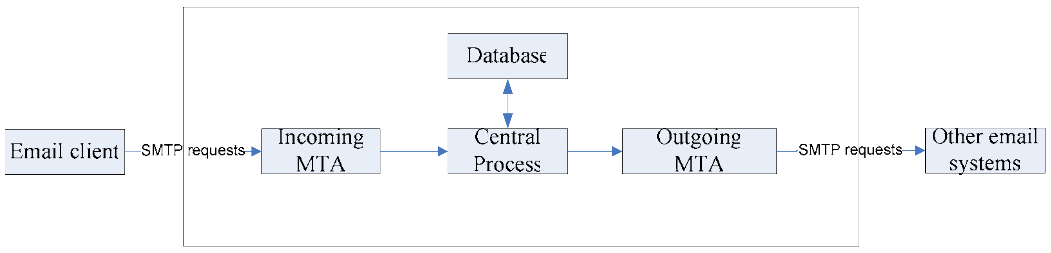


Figure 1.1: An example of a message-flow Internet service

Figure 1.1 shows a message flow of an email message that goes through an email system: At first, the message is sent from an email client to an incoming MTA (Mail Transfer Agent) via the SMTP port. Then the message will be routed though the system and eventually forwarded to third-party email systems via the outgoing MTA.

As Internet services become more and more mission-critical, ideally they are expected to be running continuously 24/7 without any problems. Any downtime will not only cost a great deal in the form of lost reputation of the service but also bring possible penalties for failing to meet service agreements with the customer. Therefore, large-scale Internet services such as Blackberry Internet Service (BIS) Engine, a popular wireless email service developed by Research In Motion, requires continuous monitoring. Hence, the developers and administrators need some mechanisms to trace the flow of messages in order to monitor the status of the system.

In general, there are several types of information and resources that can be used to monitor a system, including trace information, event log files, or special metrics such as SNMP reports. Among these resources, the event log contains the most straightforward information about the system because the log text is expected to be human readable and understandable.

In other words, event logs can help developers and administrators understand the behaviors in a system in a straightforward way. Thus, log analysis is a very important approach to keep track of the system status, as well as to determine the root causes of problems. However, due to the large amount of log data generated by dynamic Internet services, it is very time-consuming to find out useful information in the event log. In order to reduce the complexity of log analysis, some frameworks have been invented to extract valuable information from event logs and thereby detect the faults.

1.1 Motivation

Current approaches for log analysis can be categorized into the following two categories.

The first category is expert system that builds static dependency models on rules and patterns defined by human experts. In other words, problems are determined in a static manner. The IBM Log and Trace tool [1] converts syslog entries to the standard CBE [23] format and performs problem determination by matching the keywords in the log entries with the records in the symptom databases predefined by IBM experts. The correlation method among log records can be based on Timestamp, URL, and Application ID. A combination of these criteria or any user-defined criteria can also be used to correlate log entries. However, the symptom databases only contain the symptoms for IBM software products. In other words, this solution cannot be applied to the software products from other vendors.

A similar approach has been also applied to SNMP (Simple Network Management Protocol [20]) with some useful visualization [21]. In this paper, events are correlated by the rules defined by human experts. For example, a keyword such as “network card failure” should be correlated with the error “interface failure” because it is a pattern observed by system administrators.

Other frameworks such as Swatch [5] and SEC [6] require human experts to input extensive knowledge about the system. These frameworks use regular expressions to extract useful information from log entries and provide a configuration language to define the expected message flows or code paths. Hence, this type of static approach is not only limited to the knowledge that the experts understand about the system but is also vulnerable to the changes in the system.

The second category of approach is to perform syslog analysis based on statistical models. For instance, in [7][8] the authors use a text clustering algorithm instead of regular expressions to automate message typing. However, these approaches mainly focus on log event clustering and do not provide any solutions for problem determination.

Naïve Bays algorithm and hidden Markov Model have been used in the paper [9][12] to discover temporal patterns without predefined time windows. A similar approach in paper [11] is to find periodic temporal patterns using a chi-square test. On the contrary, the paper [14] correlates event logs within a predefined time window. The major limitation of these approaches is that behaviors in these systems are expected to occur periodically. For instance, a pattern consists of threshold violated and threshold reset events that occur every 30 seconds may be indicative of hosts nearing their capacity limits [14]. Due to the dynamic nature of message-flow Internet services, this condition is hard to be satisfied in every Internet service.

DNS traffic and query logs have been statistically investigated in [13] to detect spam emails. However, due to the simplicity of DNS query logs, this paper does not look into the different states or code paths of the log data. In addition, since the algorithm used in this paper is highly coupled with DNS data and related parameters, it cannot be applied to other protocols or systems.

X-Trace is an interesting framework that traces message flows, though it does not focus on the statistical analysis. Instead, it provides a message-tracing framework that injects metadata into network packets in order to trace how a particular request initialized by either clients or administrators flows through the system. If any of the expected components are missing in the flow, the missing components will be considered as faulty.

Pinpoint is by far the most similar approach to our solution. It dynamically traces real client requests going through a message-flow system and uses data clustering and statistical techniques to correlate failures of the requests to the components most likely to have caused them [3]. However, this approach is built on top of the J2EE platform. In such a case, components such as EJB, JSP pages or JSP tags are completely controlled and defined by the J2EE application server, so developers do not have any control over the granularity in statistical analysis. Furthermore, whether a request is successful or not in the Pinpoint framework is detected by Java exceptions. These two limitations prevent the Pinpoint framework from being a generic solution for common message-flow Internet services.

In sum, none of the existing problem determination frameworks provides a completely generic solution that meets the following requirements for message-flow Internet services.

1. The solution should be generic and independent of any OS or middleware framework.
2. The solution should be completely dynamic and can adapt to the evolution of Internet services.
3. The log correlation could be based on the nature of message-flow Internet services.
4. Little or no prior knowledge about the system is required.
5. There is no need to know how to determine if a message succeeds or fails.
6. Code paths, message flows, and states in the system could be completely controlled by the application developers instead of the middleware application servers.

1.2 Contributions

This paper has the following five contributions.

1. This thesis explores new ways to perform log analysis by removing the dependency on middleware platforms such as J2EE or .NET. We define a generic log adaptor to convert event logs to into our event format. The log adaptor is independent on any OS or framework. Though the CBE [23] already defines a similar event structure, we find that most fields in CBE format and the complex grammar of CBE are not necessary for a message-flow Internet service. Therefore, we create a lightweight event structure and develop a log adaptor that uses regular expressions to extract information from event logs. Unnecessary log entries can be also filtered out by customized filters.
2. We develop an algorithm of log event clustering that combines a text matching technique with the modified version of SLCT [7] [8], a text clustering algorithm. The log entries in our research contain more vocabularies and a more complicated grammar if compared to the log data in other papers.
3. We develop an algorithm of log pattern clustering by correlating log events with the same message ID. Clustered log patterns are divided into groups based on the similarities among log patterns. Since each group of log patterns represents a system behavior, we can build a precise statistical model of the system based on log patterns.
4. We introduce a sliding window technique into the chi-square goodness of fit test in order to detect problems. Our sliding window algorithm significantly reduces the number of false alarms in our test. Two types of metrics, including the number of

patterns and the processing time of each message, are used in the chi-square goodness of fit test to compare the data distribution in the current window against the history data in the baseline. The test results show a high accuracy.

5. Other than the event log format, no prior knowledge about the system is required. Furthermore, as the system evolves, e.g., upgrading to newer versions or changing log formats, the statistical model can be rebuilt automatically without any human involvement. Our assumption is that if the event logs can provide enough human readable information for experts to investigate problems manually, then it also provides enough input for our problem determination framework.

1.3 Organization

The remainder of this thesis is structured as follows. In chapter 2, we introduce the background of our work and explain how to find out the root causes of a problem in the BIS Engine. Afterwards we review three typical approaches for problem determination: LTA, X-Trace, and Pinpoint. At the end, we compare the differences between them and our framework. In chapter 3, we demonstrate the architecture of our framework and explain the algorithms of how to perform clustering of log events and log patterns. We also discuss the details of our sliding window algorithm of how to shift or freeze the sliding window. Eventually we present the process of how to determine problems with the chi-square goodness of fit test. Chapter 4 describes how we evaluate our framework with different log data sets collected from the BIS engine. In the final chapter, we summarize our solution of problem determination for message-flow Internet services and discuss some possible future work.

Chapter 2

Background and Related Work

In the previous chapter, we discuss some papers about log analysis and identify two major types of solutions: expert systems and statistical models. In the background section, we use the BIS engine system as an example to demonstrate how to perform log analysis manually to identify the root cause of a problem in a message-flow Internet service. Then we focus on three typical approaches of problem determination. Section 2.1 gives a brief introduction about LTA (Log Trace Analyzer). In section 2.2, we introduce the basic idea of the X-Trace framework. In the last section, we will give a detailed analysis of the Pinpoint framework because it is the most similar solution to our approach. At last, we compare the differences between these systems and our framework.

2.1 Background: Log Analysis in Message-flow Internet Services

The focus of this thesis is problem determination: detecting problems and identifying root causes based on event logs. In a message-flow Internet service, one of the most important methods to monitor system status is to analyze the event log since it contains the most straightforward information, human readable text. In general, for each message comes into an Internet service, the important states and the basic flow of the message will be logged in the event logs. However, current approaches do not address the problem that applications can spread debugging, warning and error information across multiple log lines in one or multiple log files. For instance, if a message routes through three components (Here a component can be anything, e.g., a process, a thread, a Java class or a C++ library), it is very common that each component writes its own information about the same message to the event logs. Hence, there will be at least three log lines related to the same message to be logged across one or multiple log files. From an Internet service's point of view, such a sequence of log lines could be considered as a log pattern because each sequence could represent a certain type of system behavior. Handling such multi-line log events requires the ability to automate the log message types.

Let us use the BIS[4] engine as an example. As shown in Figure 1.1, the BIS system is a distributed system composed of multiple components running on multiple machines.

Whenever an email message comes into the BIS system via the inbound SMTP server, a unique message ID will be generated and assigned to that particular message. The message then transmits to the CentralProcess, the database application server and eventually hits the out-bound MTA where the message is forwarded to third party email systems. The metadata containing the message ID will be forwarded to all components in the system until it leaves the system. In each component, developers write debugging logs that contain a message ID field using the standard syslog mechanism on Linux. Therefore, the log lines related to the same message ID can be retrieved with the following Linux command.

```
grep $message_id $log_file_name
```

Figure 2.1: Linux command to search for log lines related to a particular message ID

Imagining that we receive a phone call from a customer complaining that his or her email service is not working, and we find that the message ID assigned to the last message for this user is 65530, as shown in the first log line in Figure 2.2. In order to collect more detailed information about the problem, system administrators search for all log lines related to this particular message ID 65530 by running the Linux command “grep 65530 event.log” in the log file directory. The corresponding search results in Figure 2.2 show that the sequential log lines represent important states of the message and related debugging information such as user names and session data. According to the last log line, we can confirm that this error is due to a broken database connection, and therefore we can make corresponding actions to fix the problem.

```
Nov  6 15:21:48 Data,3034209200,65530,ReceiverTask::ReceivedData,"man_id=544243646,srp_id
=TEST,mailhost_id=1,sender=HTTP_SERVER,sender_addr=,receivertask_qsize=0"
…
Nov  6 15:21:48 CentralProcess1 [9386]: DEBUG,Directory processing
time,33926064,65530,ConnectorManager::svc,"Directory processing time, user_id=544243646, srp_id=test,
directory_lookup_time=0 msec, avg=0 msec, queue_time=0 msec, proxy=localhost"
…
Nov  6 15:21:48 CentralProcess1 [18478]: WARNING, database_lookup_failed, 1409307728, 65530, ,”
database lookup failed, userid=544243646”
```

Figure 2.2: Sample log lines correlated by message ID 65530

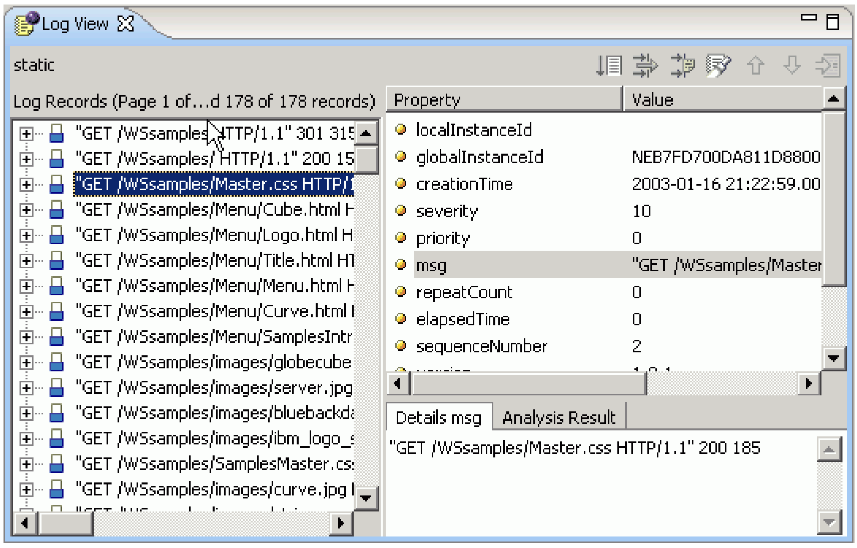
These log entries in BIS contain more vocabularies and more complicated grammar of log text when compared to the log data in other papers. More importantly, every log entry is required to contain a message ID. None of the current approaches has ever addressed problem determination based on multi-line log events that can be correlated by unique message IDs. Hence, the purpose of this thesis is to resolve this problem. In the next three sections, we will discuss three popular log analysis frameworks and explain why they do not meet the requirements for problem determination in the message-flow Internet services like the BIS engine.

2.2 IBM Log and Trace Analyzer (LTA)

LTA is an Eclipse-based tool that enables viewing, analysis, and correlation of log files generated by IBM WebSphere Application Server, IBM HTTP Server, IBM DB2 Universal Database, and Apache HTTP Server[1]. The LTA is built based on Java and Eclipse environment and it actually includes two major components: a log analysis tool and a profiling tool.

2.2.1 Event Parsing and Collection

The log analysis tool allows us to import various log files. The formats of the original log data may vary depending on the OS and software product version. For instance, Apache HTTP server uses the standard syslog while the log data of IBM DB2 database server is in its own format. Hence, all log entries need to be converted to a standard Common Base Event (CBE) format for further analysis. In fact, LTA uses an XML-base format to describe the log events and the relationships between events.



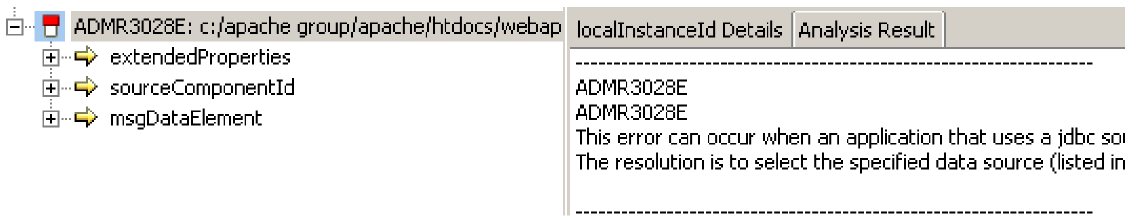
```
Log Records (Page 1 of...d 178 of 178 records)
"GET /WSsamples HTTP/1.1" 301 315
"GET /WSsamples/ HTTP/1.1" 200 15
"GET /WSsamples/Master.css HTTP/1
"GET /WSsamples/Menu/Cube.html H
"GET /WSsamples/Menu/Logo.html H
"GET /WSsamples/Menu/Title.html HT
"GET /WSsamples/Menu/Menu.html H
"GET /WSsamples/Menu/Curve.html I
"GET /WSsamples/Menu/SamplesIntr
"GET /WSsamples/images/globecube
"GET /WSsamples/images/server.jpg
"GET /WSsamples/images/bluebackd
"GET /WSsamples/images/ibm_logo_
"GET /WSsamples/SamplesMaster.cs
"GET /WSsamples/images/curve.jpg I
```

Property	Value
localInstanceId	
globalInstanceId	NEB7FD700DA811D8800
creationTime	2003-01-16 21:22:59.00
severity	10
priority	0
msg	"GET /WSsamples/Master
repeatCount	0
elapsedTime	0
sequenceNumber	2

```
"GET /WSsamples/Master.css HTTP/1.1" 200 185
```

Figure 2.3: Log views [1]

Figure 2.3 shows the GUI of the log importer. The property field on the right side of the window represents the attributes in CBE format.



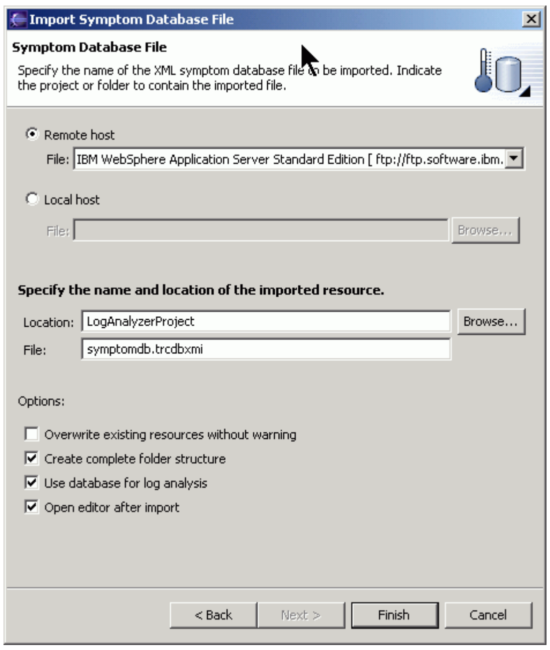
```
ADMR3028E: c:/apache group/apache/htdocs/webap
  extendedProperties
  sourceComponentId
  msgDataElement
```

localInstanceId Details | Analysis Result

```
-----------------------------------------------------------------
ADMR3028E
ADMR3028E
This error can occur when an application that uses a jdbc so
The resolution is to select the specified data source (listed in
-----------------------------------------------------------------
```

Figure 2.4: Log analysis results [1]

In order to find the problems in the log data, the LTA compares the keywords in the log lines against the predefined symbols in symptom databases. If the match for a particular keyword is found in the symptom databases, corresponding analysis results will be displayed on the GUI, as illustrated in Figure 2.4.



Import Symptom Database File

Symptom Database File

Specify the name of the XML symptom database file to be imported. Indicate the project or folder to contain the imported file.

(●) Remote host

File: IBM WebSphere Application Server Standard Edition [ftp://ftp.software.ibm.

( ) Local host

File: Browse...

Specify the name and location of the imported resource.

Location: LogAnalyzerProject Browse...

File: symptomdb.trcdbxmi

Options:

- [ ] Overwrite existing resources without warning
- [x] Create complete folder structure
- [x] Use database for log analysis
- [x] Open editor after import

< Back | Next > | Finish | Cancel

Figure 2.5: Import symptom database file [1]

IBM also provides many symptom databases for different IBM software products. Therefore, we can import symptom databases that can be loaded to analyze and correlate log files. As shown in Figure 2.5, two input sources are available, including the IBM FTP site and the local machine.

The log records can be also filtered based on the CBE properties. We can create customized filters to exam the log entries. Finding and sorting the log entries can be done in the GUI.

2.2.2 Event Correlation

The log records in LTA can be correlated by time, URLs, or application IDs. There are actually two categories of correlations. The first one is log interaction, e.g., ordering log records by timestamp. The second correlation is log thread correlation. That is to say, the log data generated by the same thread will be correlated together. Figure 2.6 depicts how two log records with the same timestamp are correlated together.

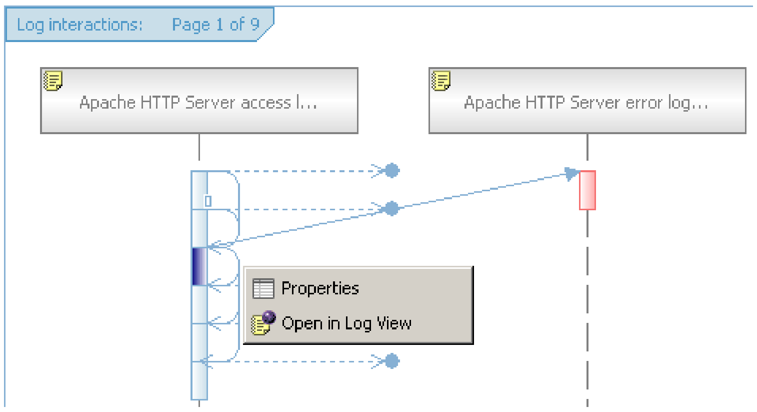

Log interactions: Page 1 of 9
Apache HTTP Server access l...
Apache HTTP Server error log...
Properties
Open in Log View

Figure 2.6: Log records that have the same timestamp [1]

2.2.3 Limitation and Restrictions

As discussed in the previous section, the LTA is actually an expert system that detects problems based on the keywords in the symptom database. Once the system evolves, the system administrators need to find an updated version of the symptom database in order to match the new keywords in the event logs. Furthermore, the LTA only provides symptom database for IBM products, so it is not possible to use this framework in third-party systems. The last limitation of LTA is that the log correlations cannot be easily customized.

2.3 X-Trace

X-Trace is a tracing framework that provides a comprehensive view for systems that adopt it [2]. Since Internet services are generally composed of distributed components such as web servers, application servers and database servers, it is always difficult to trace the flows of messages that travel through the components. The major purpose of the X-Trace framework is to resolve the tracing issue.

2.3.1 X-Trace Framework

After administrators invoke the tracing feature, the X-Trace framework injects metadata with a unique task identifier into every network packet. The metadata is forwarded to the next component or lower layers through protocol interfaces. If multiple tasks are created when a request comes to the system, then they must be constructed in a recursive manner. Since all related operations are tagged with the same task ID, eventually these tasks will create a resulting task tree.

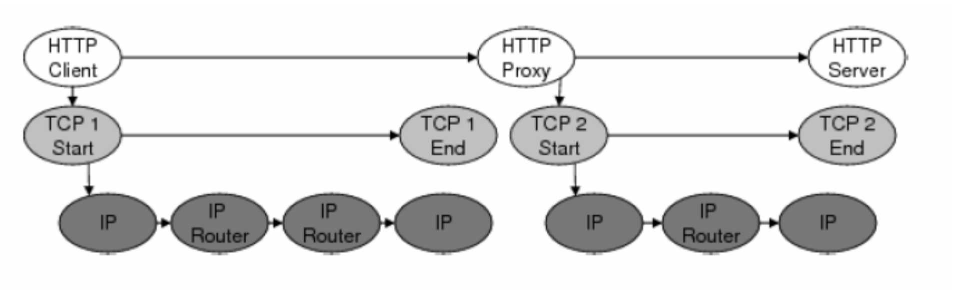


Figure 2.7: A proxied HTTP request and the logical causal relations among network elements visited [2]

Figure 2.7 illustrates an example of the task tree that reveals the relationship among application layer, TCP layer, and IP layer involved by a HTTP request. If the X-Trace framework is applied to all these layers and components, then no prior knowledge about the dependencies is required.

In X-Trace, the tracing metadata is composed of a task ID field and an optional TreeInfo field. The TreeInfo consists of three variables: ParentID, OpID and EdgeType [2]. The parent

ID is used to identify the previous layer from which the metadata is passed. The OpID refers to the operation type. The TaskID, ParentID and OpID together form a primary key.

Two functions, pushDown() and pushNext(), are used to propagate the tasks along a message path. The purpose of the pushDown() function is to copy metadata from one layer to the lower layer. The pushNext() function is used to propagate X-Tree data to the next component in the same layer. Figure 2.8 demonstrates an example of how the propagation works for a HTTP request.

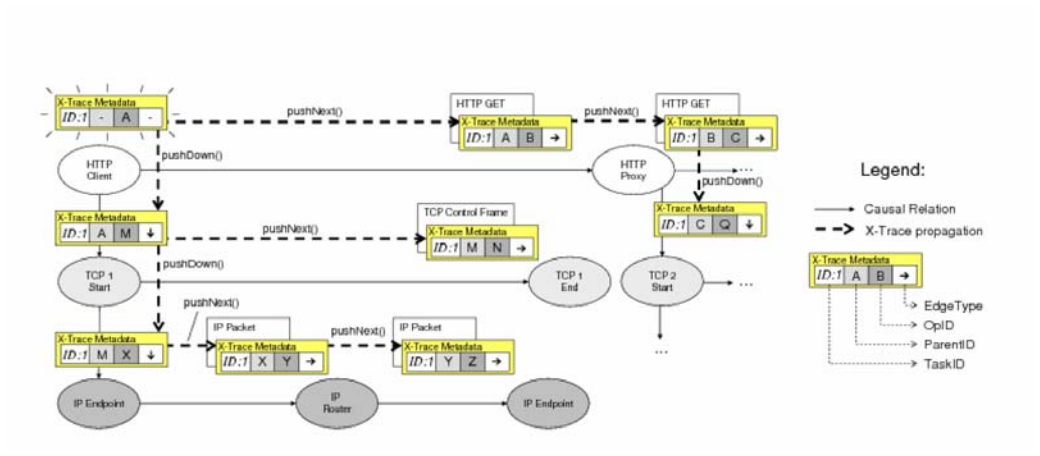


Figure 2.8: An example of propagation of metadata for a HTTP request[2]

When a node in the path receives a message that contains metadata, it generates a report and sends it to the X-Trace analysis center. The report contains a timestamp, a TaskID, and optional metadata. As illustrated in Figure 2.9, the X-Trace server reconstructs the resulting tree based on these reports.

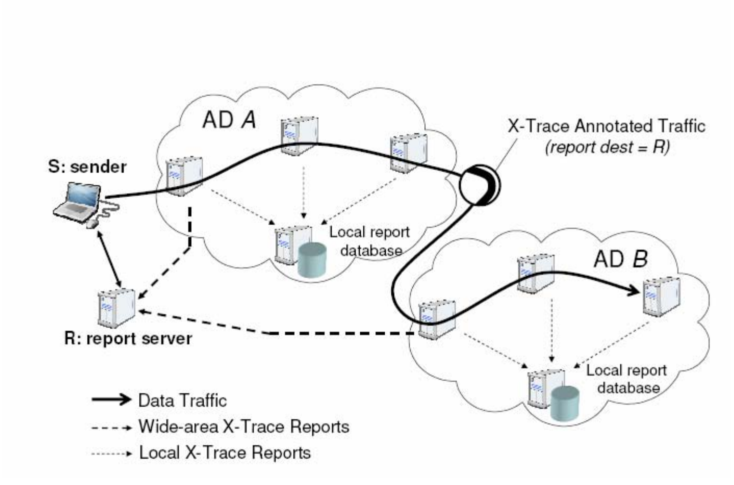


Figure 2.9: An example of how the reports are generated [2]

After the X-Trace framework is deployed, it builds a tree based on the X-Trace reports sent from all components. Once such a tree is completed, it can be used as a baseline to determine problems in the system. Let us use a website as an example to demonstrate how the X-Trace framework works. In general, a website is composed of a front-end web server and a backend database server. Hence, the problem can occur either in the web server or in the database. The fault we consider in this scenario is a wrong parameter for the database connection in the web server's configuration file. When a HTTP request comes to the web server, the metadata will not be propagated to the database because the connection to the database server cannot be established due to the wrong parameter. Thus, the X-Trace center receives only one report from the web server. On the contrary, if the database connection is established, successful requests should contain the reports from both the web server and the database server. Since we are expecting two reports, the missing report indicates a possible problem in the database.

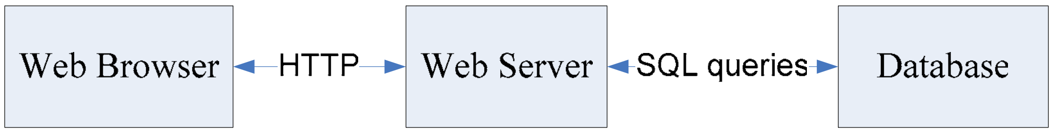


Figure 2.10: An example of a website

2.3.2 Limitations and Restrictions

The first limitation of X-Trace is that components on the path must take the responsibility of propagating the metadata. In other words, we need to modify the source code of the existing components in order to add the X-Trace metadata into the network packets.

Second, enabling the X-Trace framework will certainly have some impact on the performance of the system because it adds extra metadata into every network packet.

Third, the message-flow tree constructed in X-Trace only represents the relationships among components. That means this framework can only detect faulty components but fails to identify the actual root cause of the problem.

Last, the X-Trace framework does not create any statistical models. Instead, it provides some visualization diagrams of the network status for administrators and developers to identify successful and unsuccessful requests. In other words, human experts are required to be involved in this progress.

2.4 Pinpoint

Pinpoint is the most similar framework to our approach, so we give a very intensive analysis of this framework in this section. We also discuss the drawback and limitations of the Pinpoint framework and explain why it is not a generic solution for all Internet services.

2.4.1 The Pinpoint Framework

Pinpoint is a framework for root cause analysis on the J2EE platform that requires no knowledge of the application components [3]. Since most of the root cause analysis techniques are based on static dependency models built by human experts, the Pinpoint paper propose a new framework that handles the problem determination based on a statistical model.

The major functionalities of Pinpoint include the following two parts:

1) It allows us to trace client requests that pass through the entire system. In fact, the request concept in Pinpoint is the same as the ones in X-Trace and our framework. Tracing message flows can help us build a statistical model about the system while the static dependency model cannot.

2) It also performs data clustering and uses statistical techniques to correlate the failures of request to the components most likely to have caused those failures [3]. By correlating the failures and components, Pinpoint attempts to find the combinations of components that are believed to cause the problems.

The Pinpoint framework makes two important assumptions. The first one is that the requests in the system must be balanced in the different combinations of the components. In other words, if multiple components are always related to the same fault, then there is no way to distinguish these components for this particular fault. The second assumption is that the requests are independent to each other in case of failures. That is to say, failed requests should not be caused by the activities of other requests. The Pinpoint framework can be represented in the following figure.

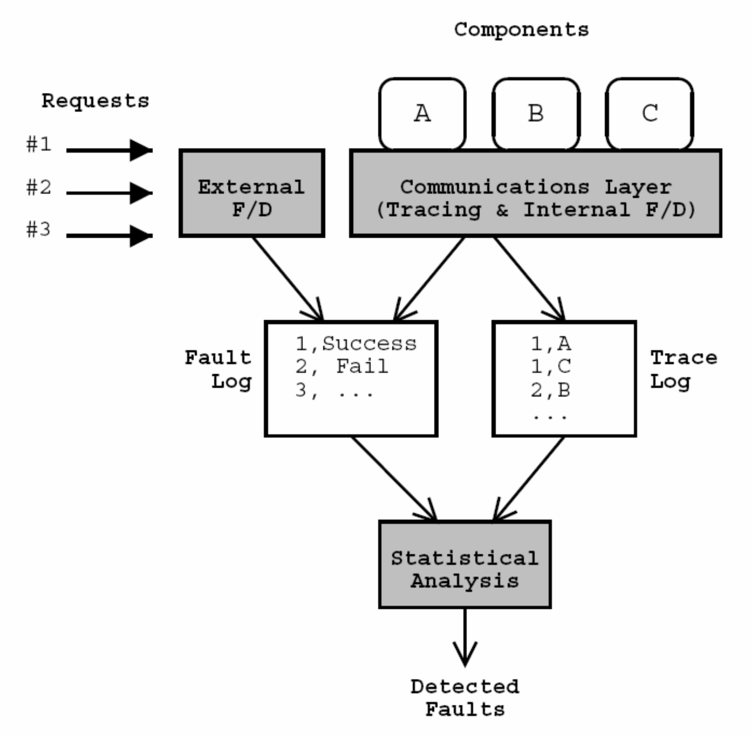


Figure 2.11: The Pinpoint framework [3]

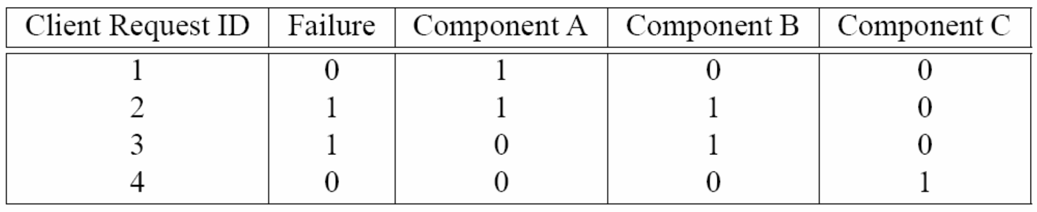
The basic idea of how to trace a message in Pinpoint is to store all the components that a message uses when it routes through the system. Since the Pinpoint relies on the J2EE application server to keep track of the message flow among components, the information that Pinpoint can use is completely controlled by the J2EE platform. In general, components in

Pinpoint include software components, EJB (Enterprise Java Bean), component versions, or even database files.

A request ID will be assigned to each request by the J2EE application server. The request ID will be forwarded to the next component by the J2EE framework. This is an advantage of the Pinpoint framework, as it is not required to modify the application-level code to propagate the metadata as what X-Trace needs. In the current implementation of Pinpoint, the request ID is actually stored in thread-specific local state.

The Pinpoint uses the Java exceptions to determine failures. It tries to catch exceptions as more as possible in order to find out the root cause of a problem that might be masked by the other high-level errors. There are two types of failures in Pinpoint. The first type of failure is internal failure, which means the failure might be masked and invisible to end users. The second one is called external failure, as it is visible to end users.

Once the request traces and related logs are collected and analyzed, an input matrix, shown in following table, can be generated for further data analysis.



Client Request ID	Failure	Component A	Component B	Component C
1	0	1	0	0
2	1	1	1	0
3	1	0	1	0
4	0	0	0	1

Table 2.1: A sample input matrix for data analysis [3]

Pinpoint uses a data clustering algorithm called UPGMA to group similar data points together in order to detect whether the client requests are successful or not.

The Pinpoint framework is validated with the J2EE PetStore demonstration application by manually injecting different types of faults into the system. The testing results show good accuracy.

2.4.2 Limitations

The major limitation of Pinpoint is that it cannot distinguish between sets of components that are tightly coupled and are always used together [3]. On the contrary, the components in our framework can be completely independent if the event logs are correctly written by the developers.

The second major limitation is that Pinpoint is built based on the J2EE platform, so it is not a generic solution for all Internet services. For instance, many open source projects such as MySQL and Apache are not implemented on top of J2EE, so Pinpoint cannot be applied to these systems.

Pinpoint also relies on the Java exception handler to catch failures. Therefore, if the system behavior is changing dramatically but no exception is thrown, e.g., high CPU usage due to DDOS attack, Pinpoint will fail to detect the significant change in the system immediately.

Last, Pinpoint focuses on the component level, so it does not know about the internal states of each component. Given the increasing complexities of large, dynamic Internet services, a finer granularity would be necessary for developers and administrators to analyze the root causes of problems.

2.5 Comparison

In the previous sections, we have discussed three popular solutions for problem determination. In this section, we use the following table to illustrate the major differences between these solutions and our framework.

	LTA	X-Trace	Pinpoint	Our framework
Model (Category)	Expert system	Expert system based on visualization diagrams	Statistical model built based on UPGMA and Jaccard similarity coefficient	Statistical model built based on the chi-square goodness of fit test with a sliding window approach
Input source	Traces, event logs collected from IBM products	Metadata injected by X-Trace	Trace information provided by the J2EE platform	Text log files
Root cause analysis	Symptom databases	Identified by human experts based on visualizations of network diagrams	Failures detected by Java exceptions	Detect changes in the system by comparing test log data against baseline
Log correlation	Application ID, timestamp, or thread ID	Client request ID is assigned at the network protocol level by the X-Trace framework	Client request ID assigned by the J2EE application server.	Correlate the log records by the message ID logged in every log line.

Component granularities	Controlled by log records and symptom databases	Network level, e.g., machine names or host names	J2EE components such as EJB, JSP and JSP tags	At log record level. The more details the application developers put into the event logs, the more precise the model is

Table 2.2: Difference among the three solutions and our framework

Chapter 3

Problem Determination

In this chapter, we will describe the details of our problem determination framework. At first we briefly introduce the architecture of our framework and discuss how the components in our framework work together to perform problem determination. Then we will focus on the algorithms of log event clustering and log pattern clustering. Eventually we will explain how to detect problem using the chi-square goodness of fit test and the corresponding sliding window algorithm.

3.1 Log Analysis Framework

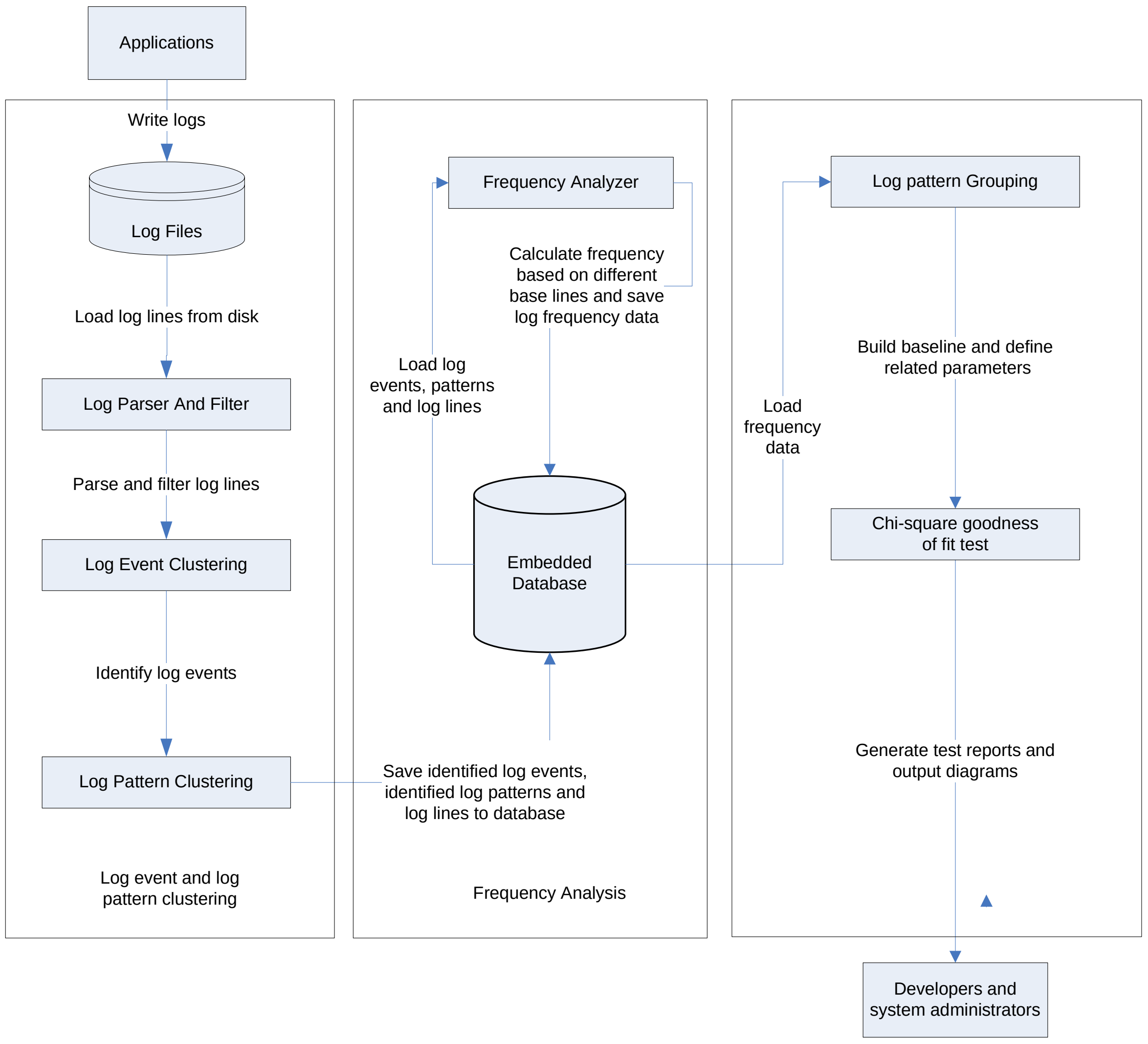

Applications
Write logs
Log Files
Load log lines from disk
Log Parser And Filter
Parse and filter log lines
Log Event Clustering
Identify log events
Log Pattern Clustering
Log event and log pattern clustering
Frequency Analyzer
Calculate frequency based on different base lines and save log frequency data
Load log events, patterns and log lines
Embedded Database
Save identified log events, identified log patterns and log lines to database
Frequency Analysis
Log pattern Grouping
Build baseline and define related parameters
Load frequency data
Chi-square goodness of fit test
Generate test reports and output diagrams
Developers and system administrators

Figure 3.1: System architecture of our problem determination

As shown in Figure 3.1, our framework is composed of the following components:

1) Input

The input of our framework is log file in text format. We do not define the process of how to collect log files because it is not our concern here. The only requirement about the log format is that a message ID should be logged in each log line.

2) Log event and pattern clustering

The next step is to identify the clustering of log events and log patterns in the log data. First, we use regular expressions to parse log entries in order to extract useful information from the log text. Log entries can be also filtered by customized filters. We develop a text-matching algorithm and a text-clustering algorithm to identify log events. The log pattern clustering can be done by correlating the log events with the same message ID. Eventually, the classified log events, patterns, and log text will be inserted into an embedded database (SQLite) for further analysis.

3) Frequency analysis

In this step, we calculate the frequency of log patterns in each time window by running some complex SQL queries on the SQLite database. We are interested in two types of data: the number of log patterns and the processing time of each log pattern. Eventually we can build a statistical model that describes the statistics of behaviors in the system.

4) Testing

The results of frequency analysis will be used as input for our chi-square test. We run a chi-square goodness of fit test to determine if the new event logs fit the statistical model in the baseline. We also develop a sliding window algorithm that does not only fit the trend of dynamic workload of Internet service but also significantly reduces the number of false alarms.

5) Output

The output of our framework would be summary tables and diagrams about abnormal behaviors detected in the system. If the diagrams and tables show a significant change in the system, developers or administrators could look into the corresponding abnormal log patterns to find out the root cause.

In the following sections, we will discuss the details of the above steps one by one.

3.2 Log Record

A log file may contain multiple log lines. From a log analysis point of view, each log line can be regarded as a log record. Since the smallest granularity of data in our framework is the log line, all algorithms and operations would be performed at the log line level.

3.2.1 Log Format

In general, event logs contain a variety of information about the system status, including events from various applications, processes, as well as user requests such as incoming or outgoing messages. Furthermore, event logs are usually written in a certain format, e.g., a timestamp when the event occurs following by a readable text string that describes the actual event. The syslog RFC [26] requires a log record to contain a timestamp, a string identifying the source of the message, and a free-format 1024-byte ASCII description of the event. In the BIS engine, all types of event logs, including debugging information, warnings, or critical errors, are logged using the standard syslog mechanism on Linux. Below is a sample log line in the BIS engine.

```
Sep 20 03:25:00 host1 CentralProcess1 [9386]: DEBUG, receive status, 3006249904, 65538,
ReceiverTask::ReceivedStatus,"status=COMPLETED, sender_addr=127.0.0.1:40877"
```

Figure 3.2: A log line example in BIS

The log line in Figure 3.2 indicates that a component named CentralProcess1 running on host1 received a COMPLETED status from 127.0.0.1 at 3:25, Sep 20. Further log analysis shows that the event log format used in BIS can be described in the following table:

Field	**Value**	**Description**
Timestamp	Jun 2 15:00:15	The timestamp when the event occurs
Hostname	host1	The host machine on which the process is running
Component	CentralProcess1	The name of the application
Process ID	[20315]	A unique ID to identify a process in a Unix or Linux system
Log Text	DEBUG, receive status, 3006249904, 65538, ReceiverTask::ReceivedStatus,"status=COMPLETED, sender_addr=127.0.0.1:40877"	1024-bytes ASCII descriptions of the event
Log Severity	DEBUG	Log severity level
Message ID	65538	A unique integer to identify a message in the BIS engine

Table 3.1: Syslog log format used in BIS

- Log Severity

The log severities field used in BIS follows the standard syslog protocol. Table 3.2 describes the error codes and corresponding explanations.

Numerical code	Text code	Severity
0	EMERGENCY	Emergency: system is unusable

1	ALERT	Alert: action must be taken immediately
2	CRITICAL	Critical: critical conditions
3	ERROR	Error: error conditions
4	WARNING	Warning: warning conditions
5	NOTICE	Notice: normal but significant condition
6	INFO	Informational: informational messages
7	DEBUG	Debug: debugging messages

Table 3.2: Log severity information [26]

- Log Text

The log text field can contain any number of user-defined elements as long as the length is equal to or less than 1024 bytes. Furthermore, a message ID is required to be logged in the log text field so that multiple log lines can be correlated.

3.3 Log Parser and Filter

The only prior knowledge about the system required by our framework is the log format. Once the format is determined, information can be extracted from the log records and unnecessary data can be filtered out. For the BIS engine, we use the following regular expressions to perform log parsing and filtering.

Field	Regular expression
TIME_STAMP	(.{3}\\s*.*?\\s.*?)\\s.*?[\\d*].*
HOST_NAME	.{3}\\s*.*?\\s.*?\\s(.*?)\\s.*?[\\d*].*
COMPONENT_NAME	.{3}\\s*.*?\\s.*?\\s.*?\\s(.*?)\\[\\d*\\].*
LOG_LEVEL	.{3}\\s*.*?\\s.*?\\s.*?\\s(.*?)\\[\\d*\\]:\\s(.*?),.*
LOG_TEXT	.*?\\[\\d*\\]:\\s(.*)
MESSAGE_ID	.*[\\d*]:.*,.*,.*,(\\d+),.*

Table 3.3: Regular expression rules for BIS

3.4 Log Event Clustering

3.4.1 The Concept of Log Event

The log event is one of the most important concepts in our framework. In a distributed system that generates different types of logs, it is very common that many log entries are written in a similar format. These log messages generally look very alike except that certain attributes or values are different. For instance, the only differences between the following two log lines are the values of the ip parameter and the port parameter.

```
Sep 20 03:25:00 host1 CentralProcess1 [9386]: DEBUG, receive status, 3006249904, 65538,
ReceiverTask::ReceivedStatus,"Receive COMPLETED status, ip=10.3.2.1, port=40877"
Sep 20 03:25:01 host1 CentralProcess1 [9386]: DEBUG, receive status, 3006249904, 65539,
ReceiverTask::ReceivedStatus,"Receive COMPLETE status, ip=10.3.2.2, port=40878"
```

Figure 3.3: Examples of log lines classified as the same log event

From observers' point of view, both of the log lines in Figure 3.3 represent the same event type that indicates the component CentralProcess1 receives a COMPLETED status from the sender. Therefore, they represent a cluster in the first dimension and correspond to the same log event defined in Figure 3.4. The timestamp field, the message ID field, and the values of the ip and port parameters are replaced with wildcards "*".

```
* host1 CentralProcess1[*]: DEBUG,receive status,*,*,ReceiverTask::ReceivedStatus,
"Reveived COMPLETED status, ip=*, port=*"
```

Figure 3.4: The log event represented in Figure 3.3

The reason why such type of event is generated is that log events are usually written by developers using certain string formatting functions, e.g.,

```
sprintf("%s host1 CentralProcess1[%u]: DEBUG,receive status,%u,ReceiverTask::ReceivedStatus,
"Reveived COMPLETED status, ip=%s, port=%u", strTimeStamp, nProcessID, nMessageID, strIP, nPort);
```

Figure 3.5 Log string format using the standard sprintf() function

If the same event is logged many times, we can find many similar log lines in the log files. For small log data sets, we might be able to identify the log event types by manually reviewing the log lines. However, for large log data sets, this process needs to be automated using data mining techniques. In the following subsection, we will represent a data clustering algorithm based on the nature of the string format of event log.

3.4.2 Skip Whitespace Algorithm

Our purpose is to identify log event types automatically, so we develop an algorithm called "skip whitespace" that identifies log events based on the similarities between two log lines. The process of how to perform clustering on the log examples in Figure 3.4 is as below.

1. The log data passed into the algorithm should have been already normalized, e.g., being parsed using the regular expressions described in section 3.3. Only three

fields including timestamp, the message ID, and the log text are required while other fields can be omitted.

2. For each log text field, words are split on spaces or any predefined symbols. If a word contains an equal sign '=' or any other predefined symbols, we split the word into two parts and remove the symbol and the right part. For example, the string "ip=10.3.2.1" can be divided into a left part "ip" and a right part "10.3.2.1". In such a case, the left part "ip" remains while the equal sign and the right part "10.3.2.1" will be removed.

3. For the first log line, since the log event map is empty at this time, we create a new log event based on the first log line and insert it into the log event map.

4. We continue to classify the rest of the log lines by comparing them with the existing log events. If the similarity between a new log line and any of the existing log events is higher than a predefined threshold, then the new log line should be identified as the existing log event. Otherwise, we create a new log event based on that log line.

5. The above process will be applied to all log entries.

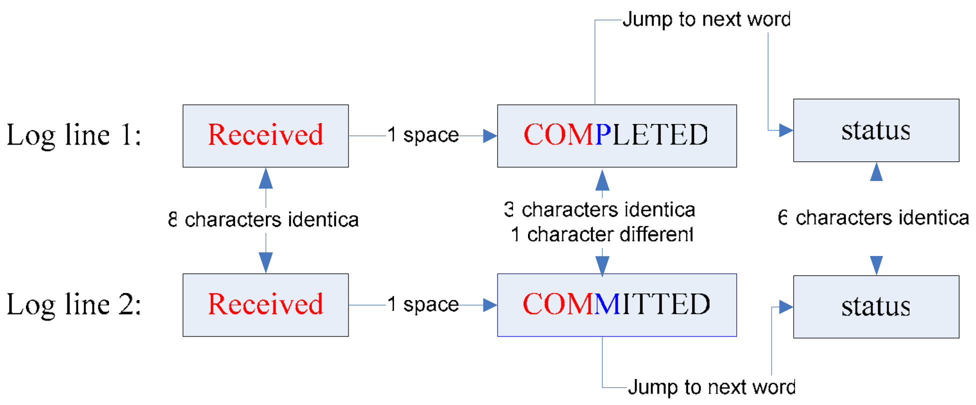


Figure 3.6: The "skip whitespace" algorithm

One thing we need to address is how to calculate the similarities between two log lines. Considering the two log lines in Figure 3.6 as examples, we compare the first character in of the first log line with the first character in the second log line. If they are identical, we increase the matching counter by one and move the current position to the next character. If they are not identical, we skip the rest characters of the word and jump to the first character of the next word. When we reach the end of either log line, we calculate the similarities between these two log lines by dividing the matching counter by the total number of characters of the first log line. If the result is larger than a predefined threshold (default value is 70), then these two log lines should be similar enough to be identified as the same log event. For instance, the similarity of the two log lines in Figure 3.6 can be calculated as below:

$$Similarity = (8 + 1 + 3 + 1 + 6) / (8 + 1 + 9 + 1 + 6) = 75\% > 70\%$$

Equation 3.1: Log event similarity

Since the result 75% is higher than the default threshold 70%, these two log lines should be identified as the same log event. When the classification is done, we will have a log event map that contains all log events identified in the training log data.

3.4.3 SLCT Algorithm

When using the above “skip whitespace” algorithm to cluster log events, we find a serious problem. The log lines in Figure 3.7 would be classified as the same event because of the high similarities among them. However, if we manually review these log lines, we can see they actually represent two different meanings: the first two log lines represent RECEIVED events while the last two lines indicate SENT events.

```
Log line 1: Received COMPLETED status, sender_addr=127.0.0.1
Log line 2: Received COMPLETED status, sender_addr=127.0.0.1
Log line 3: Sent     COMPLETED status, sender_addr=127.0.0.1
Log line 4: Sent     COMPLETED status, sender_addr=127.0.0.1
```

Figure 3.7: Event logs representing two meanings

Therefore, using log similarity is not reliable enough to identify the log events in a complex Internet service like BIS. In order to resolve this problem, we need to apply a modified version of the SLCT algorithm [7] to the log events that have been already identified by the skip-whitespace algorithm. The SLCT algorithm detects the word clusters in the log events by computing the statistical information of each word. Briefly speaking, if a word in a fixed position in multiple lines has the support of at least K (a predefined threshold), then a new word cluster is detected. The details of the modified version of the SLCT algorithm [8] can be described as below:

1. Split words on white spaces or commas in log lines that have been identified as the same event. If a word contains an equal sign, then the left part to the equal sign can be regarded as an attribute and the right part to the equal sign is considered as a value. For instance, if an input string is "job_id=545678", then the string "job_id" is the attribute while the string "546678" is the value. A set of fixed attributes are defined as $\{(n_1, v_1),...,(n_k, v_k)\}$, where n is the position of the attribute in a log line and v represents the value of the attribute. The algorithm makes a pass over all log lines belong to the same event and builds an attribute summary.
2. Then we build all possible cluster candidates by calculating the occurrence of each attribute in each position. Candidates with support higher than K (=2) will be reported as a cluster and added to a temporary cluster map.
3. After the frequency analysis for a log event is done, we check the temporary cluster map and see if it contains more than one item. If so, the original log event will be split into new events according to the clusters in the temporary map.

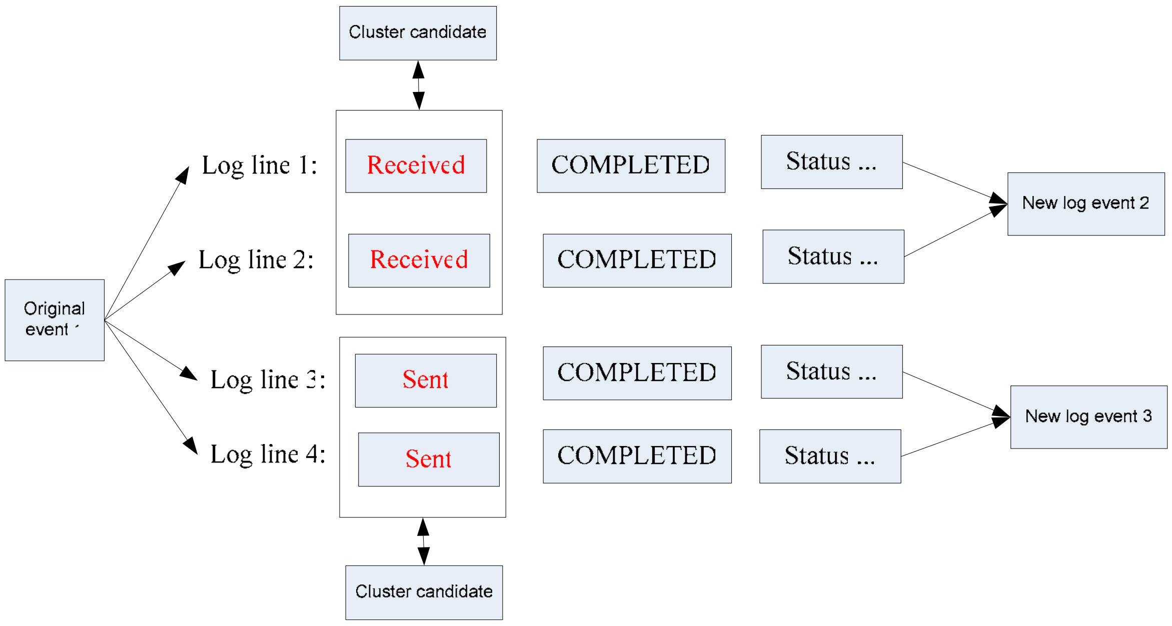


Figure 3.8: Cluster candidates identified in Figure 3.7

For instance, as shown in Figure 3.8, the words *Received* occur at the same position in the line 1 and line 2 while the world *Sent* occurs in the line 3 and line 4, so a set of fixed attributes *{(1, 'Received), (1, 'Send')}* becomes two cluster candidates. Since the number of occurrences of the word *Received* or *Sent* are both equal to predefined threshold K (K=2), the original log event 1 will be split into two new events: log event 2 and log event 3. The new log event 2 represents the original line 1 and line 2 and the new log event 3 represents the original line 3 and line 4.

Log line number	Log text	Cluster candidate keyword	Original log event ID	New log event ID
1	Received COMPLETED status, sender_addr=127.0.0.1	receive	1	2
2	Received COMPLETED status, sender_addr=127.0.0.1	receive	1	2
3	Sent COMPLETED status, sender_addr=127.0.0.1	sent	1	3
4	Sent COMPLETED status, sender_addr=127.0.0.1	sent	1	3

Table 3.4: Results of the SLCT algorithm

3.4.4 The Reasons to Combine Two Algorithms

First, as we have discussed in the previous section, the whitespace algorithm cannot distinguish the difference between two similar log lines.

On the other hand, the SLCT algorithm is not able to resolve the following problem where a word has a support of K in two log lines that actually represent two different events.

```
Log example 1:  RECEIVED COMPLETED STATUS FROM COMPONENT
Log example 2:  SENT         COMMITTED STATUS TO      COMPONENT
```

Figure 3.9: Log examples that cannot be identified by SLCT

Considering the log lines in Figure 3.9 as examples, since the cluster candidates such as *STATUS* and *COMPONENT* have occurred twice, which is equal to the predefined threshold (support K=2), the SLCT algorithm will consider these two log lines as the same log event.

```
Logevent:  * * STATUS * COMPONENT
```

Figure 3.10: A cluster identified by SLCT

However, if we manually review the log texts, it is obvious these two lines have different meanings: The first log line indicates that a status is received while the second one means a status is sent.

$$Similarity = (0 + 1 + 3 + 1 + 6 + 0 + 1 + 9) / (8 + 1 + 9 + 1 + 6 + 1 + 4 + 9) = 54\%$$

Equation 3.2: Log event similarity

Fortunately, the whitespace algorithm is able to resolve this issue because the similarity between these two log lines is lower than the predefined similarity threshold. Hence, they would not be identified as the same event by the skip-whitespace algorithm. By combining the skip-whitespace and SLCT algorithms, we would be able to overcome the disadvantage of each algorithm and create a more accurate result of log event clustering.

3.5 Log Pattern Clustering

3.5.1 Log Correlation

In expert systems, log lines are usually correlated by rules defined by experts. In our framework, we have an assumption that a unique message ID will be logged in every log line. From a message-flow system's point of view, such a message ID is very necessary for developers and administrators to trace the flow and the status of a message traveling through all components.

```
Sep 20 03:25:00 host1 CentralProcess1[9386]: DEBUG,User not in cache,3079662512,
65538,UserCache::FindUser,"User not found in cache,user_id=5371100000, state=0"

Sep 20 03:25:01 host1 CentralProcess1[9386]: DEBUG,receive status,3006249904,
65538,ReceiverTask::ReceivedStatus,"status=COMMITTED,sender_addr=127.0.0.1"

Sep 20 03:25:02 host1 CentralProcess1[9386]: DEBUG,receive status,3006249904,
65538,ReceiverTask::ReceivedStatus,"status=COMPLETED,sender_addr=127.0.0.1"
```

Figure 3.11: Examples of event logs with message IDs

As shown in Figure 3.11, three log lines related to the same message ID 65538 are displayed in a sequential order in which the log lines are logged. The message ID can be extracted using the regular expression rules defined in Table 3.3.

3.5.2 Definition of log pattern

In this thesis, a log pattern is defined as a collection of log events correlated by the same message ID. The algorithm of how to detect clusters in the log patterns will be described in this section. Since the event ID of log records has already been identified in section 3.4, we can correlate the log events with the same message ID. As we have discussed in the previous section, a log pattern is defined as below.

A log pattern = a sequence of log events correlated by the same message ID.

```
EventID=30, LogLine=Sep 20 03:25:00 host1 CentralProcess1[9386]: DEBUG,HTTP Session received
notification,3021577136,65538,HTTP::ReceiveDataCallback,"handler_id=1, session_id=1212
EventID=31, LogLine=Sep 20 03:25:00 host1 CentralProcess1[9386]: DEBUG,Received
Data,3034209200,65538,ReceiverTask::ReceivedData,"man_id=123456789,srp_id=TEST,mailhost_id=1,sende
r=HTTP_SERVER,sender_addr=,receivertask_qsize=0"
EventID=32, LogLine=Sep 20 03:25:00 host1 CentralProcess1[9386]: DEBUG,Thread pool
enqueue,3034209200,65538,JobThreadPool::Enqueue,"asyncid=0,jobtype=SEND_MESSAGE_WORK"
…
EventID=36, LogLine=Sep 20 03:25:00 host1 CentralProcess1[9386]: DEBUG,get
proxy,57060272,65538,ConnectorManager::GetProxy,"get proxy, asyncid=65538,
user_id=123456789,srp_id=TEST
EventID=37, LogLine=Sep 20 03:25:00 host1 CentralProcess1[9386]: DEBUG,Directory processing
time,33926064,65538,ConnectorManager::svc,"Directory processing time,  user_id=123456789, srp_id=TEST,
directory_lookup_time=0 msec, proxy=localhost"
….
EventID=74, LogLine=Sep 20 03:25:01 host1 CentralProcess1[9386]: DEBUG,get user
result,47610800,65538,UserManager::GetUser_Result," get user
result,asyncid=65538,user_id=123456789,srp_id=TEST,time=15 msec
```

Figure 3.12: Log events correlated by the message ID 65538

The log events correlated by the message ID **65538** are shown in Figure 3.12, which can be described as a temporary pattern 1 using the following log pattern format:

```
temp_pattern_1=30,31,32,35,36,37,38,40,42,43,44,48,54,55,56,57,58,59,60,61,62,63,65,74
temp_pattern_2=1,2,5,6,7,14,15,17,19,20,22,23,24,25,26,29,30,31,32,34,35,36,40,42,48,49,50,51,52,53,54,55,5
6
temp_pattern_3=19,20,22,23,24,25,26,29,30,31,32,34,35,36,37,38,39,49,51,52,53,54
temp_pattern_4=1,2,5,6,7,14,15,17,18,19,20,22,23,24,25,26,29,30,31,32,42,48,49,50,51,52,53,54,55,56
…
```

Figure 3.13: Sample log patterns

We apply the above log correlation algorithm to all log lines and insert all possible log pattern formats into a temporal list for further analysis. Each item in the list might contain various numbers of log event IDs, as illustrated in Figure 3.13. The number of temporal patterns should be equal to the number of unique message IDs in the log data.

3.5.3 Pattern clustering

In the next step, we need to identify unique patterns in the temporal pattern list using a matching algorithm. First, we select an item from the temporal list and compare it with the existing patterns in the log pattern map. If the item in the list is a subset of or equal to an existing item in the log pattern map, it is identified as an existing pattern. If the item is a superset of a record in the log pattern map, the existing record in the map is replaced with the item. If there is no existing pattern that matches the item, then a new pattern is created and added to the log pattern map. The procedure is subsequently applied to all items in the temporal pattern list.

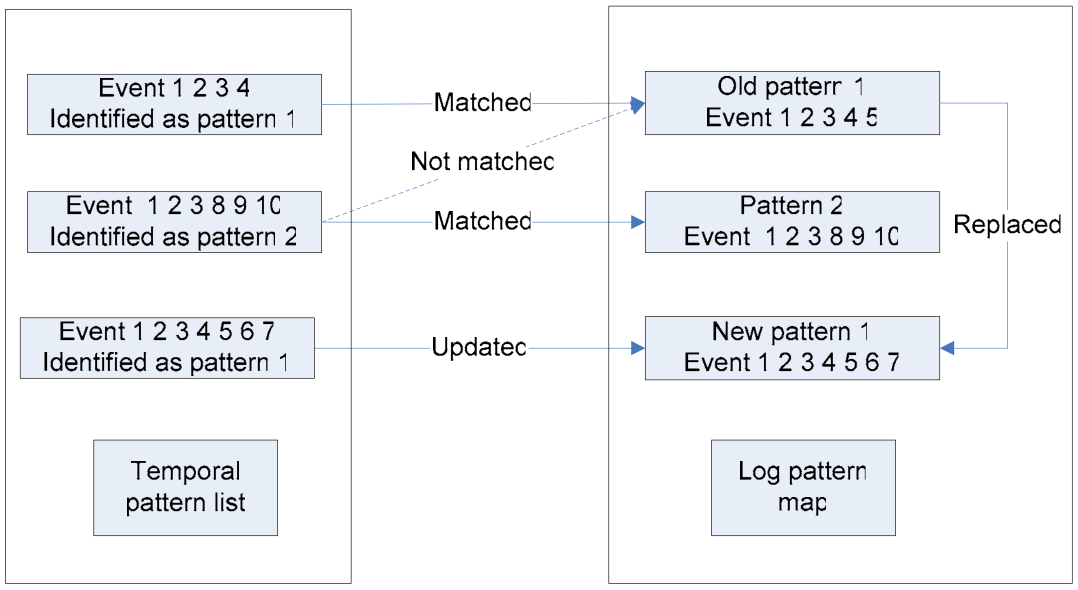


Figure 3.14: Pattern matching algorithm

An example of how the matching algorithm applies to three log lines is illustrated in Figure 3.14. Our assumption is that a log pattern 1 already exists in the pattern table. The first item in the temporal pattern list identified as pattern 1 because it is a subset of pattern 1 ({1,2,3,4} ∊ {1,2,3,4,5}). Then we compare the second item with the log pattern 1. Though the first 3 events are identical, the rest do not match ({1,2,3,8,9,10} != {1,2,3,4,5}). Therefore, we create a new pattern 2 based on item 2. Item 3 is considered as a superset of pattern 1, so the old events in the pattern 1 will be replaced with the new events in item 3.

Moreover, the sequential order in which the log entries are log logged is highly dependent on the syslog mechanism and does not always reflect the real situation, so the order of log events is ignored in our clustering algorithm. In other words, our solution does not consider partial order.

3.6 Implementation Information and Data Structures

Platform	Linux, Windows

Lines of code	12,500 lines C++
Configuration Files	2
Number of Classes	36

Table 3.5: Implementation information

Table 3.5 shows the implementation information about our log analysis framework. The project contains about 12,500 lines of C++ code and 36 C++ classes. It also supports both Linux and Windows platforms and accepts a configuration file name as an argument. Below are some important C++ classes that need to be elaborated.

- LogLine class

```
class LogLine
{
public:
    LogLine();
   ~LogLine();
   bool Parse(const std::string& strLogLine);
   bool Filter(const std::string& strLogLine);
…
private:
   int          m_nSeqID;
   int          m_nLogEventID;
   int          m_nLogPatternID;
   int          m_nMessageID;
   int          m_nLogLevel;
   int          m_nTimestamp;
   std::string             m_strComponentName;
   std::string             m_strHostName;
….
};
```

The LogLine class contains the most important variables related to the log record. For instance, the m_nMessageID variable represents the unique message ID logged in each line. The m_nLogEventID and m_nLogPatternID variables indicate the corresponding event ID and pattern ID of the particular instance of the LogLine class.

- LogEvent class

```
class LogEvent
{
public:
          LogEvent();
          ~LogEvent();
          void Initialize();
          int  DetectClusters(LogLine *pLogLine);
```

```
        int  AnalyzeWordFrequency(LogLine *pLogLine);
        int  Purify(LogPattern& clsLogPattern, SQLiteInterface& clsSQLiteInterface,
std::vector<LogLine *> &vectNewLogLine);
        int SplitEvent(const int nEventID, std::multimap<int, LogLine *> &mmapNewEvent, bool
bNeedSplit = true);
        double  GetSimilarity(LogLine *pSrcLogLine,  LogLine *pDestLogLine);
…
private:
        MULTIMAP_LOG_EVENT              m_mmapLogEvent;
        MAP_LOG_EVENT                   m_mapLogEvent;
        MAP_LOG_WORD                    m_mapLogWord;
…
};
```

Each instance of the LogEvent class represents a log event identified in the log data. The aggregation relationship between the LogEvent class and the LogLine class is because a log event can represent multiple log lines. In memory, multiple LogLine instances share the same LogEvent instance according to the log event ID.

- LogPattern class

```
class LogPattern
{
public:
        LogPattern ();
        ~LogPattern ();
        void Initialize();
        int  Classify(MAP_LOG_EVENT &mapLogEvent, SQLiteInterface& m_clsSQLiteInterface);
        bool  GetSimilarGropus(MMAP_LOG_SIMILAR_GROUP &mmapLogSimilarGroup, int
&nGroupCount);
        bool GetSimilarity(int nSrcPatternID, MAP_LOG_EVENT &mapSrcLogEvent,
                                    int nDestPatternID, MAP_LOG_EVENT &mapDestLogEvent);
        int  Compare(MAP_LOG_EVENT &mapSrcLogEvent, MAP_LOG_EVENT
&mapDestLogEvent);
…
private:
        MAP_LOG_PATTERN          m_mapLogPattern;
        int m_nLogPatternSeqID;
        std::string m_strDatabaseDir;
...
};
```

Each instance of the LogPattern class represents a unique log pattern, and thus a LogPattern instance can be referenced by one or many LogEvent instances and LogLine instances.

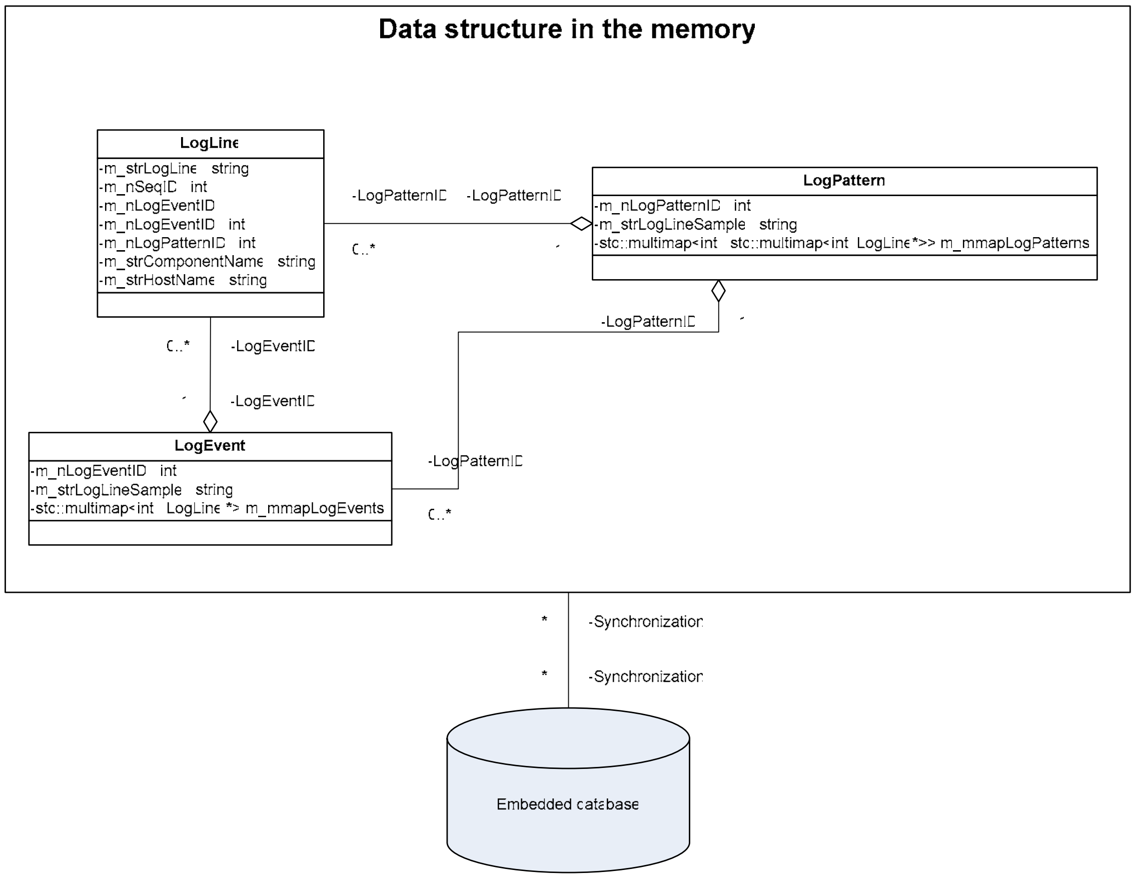


Figure 3.15: Data structures and classes

In Figure 3.15, we demonstrate the relationships among the three major classes using standard UML. The multiplicity indicator “0..*, 1” means zero or many objects can be associated with another object. The arrowhead at the end of the line indicates the direction of the association. The member functions and interfaces are omitted in the diagram. We summarize their relationships below.

- Each log event represents a unique log message type. Each instance of the LogEvent class can be referenced by one or many instances of the LogLine class
- A log pattern consists of a sequence of log events correlated by the same message ID. Each instance of the LogPattern class can be referenced by one or multiple instances of the log event class.

```
typedef std::map<int , LogLine *>  MAP_LOG_EVENT;
typedef std::map<int , MAP_LOG_EVENT > MAP_LOG_PATTERN;
```

Figure 3.16: Cascade relationship among LogLine, LogEvent, and LogPattern classes

Figure 3.16 shows a cascade relationship among these classes. For instance, whenever an old log event is split into two new events during the log event clustering, all LogLine instances referencing the old log event should be updated to reference the new log event. The same mechanism should be also used when any of the log patterns are modified. In such a case, the log event instances referencing the old log pattern should be pointed to the new log pattern instance.

In other words, the relationship among LogLine, LogEvent, and LogPattern is very similar to the cascade foreign keys in a RDBMS database system. The corresponding reference keys are log line sequence ID, log event ID and log pattern ID.

3.7 Log Pattern Grouping

After pattern clustering is finished, we find an interesting factor in the results: some of the log patterns share many common log events.

```
PatternID=1, Events=30,31,32,35,36,37,38,40,42,43,44,48,57,58,59,60,63,65,74
PatternID=2, Events=31,32,35,36,37,38,40,42,43,44,46,48,57,58,59,60,63,65,74
PatternID=3, Events=31,32,35,36,37,38,40,42,43,44,46,54,55,56,57,58,59,60,61,62,63,74,75,76
PatternID=10, Events=30,31,32,35,36,37,38,40,42,43,44,54,55,56,57,58,59,60,61,62,63,74,75,76
…….
```

Figure 3.17: An example of patterns consisted of similar events

If we take a look at the first two log lines in Figure 3.17, we can see that log pattern 4 and log pattern 5 share many identical log events such as 31, 32, 35, 36, 37, 38, 40, 42, 43, 44, 48, 57, 58, 59, 60, 63, 65, and 74. The only difference between them is that the first log pattern contains the log event 30 while the second one contains the log event 46. We are very curious to know why these patterns look alike, so we manually review the log text of each log event.

```
EventID=30, LogLine=Sep 20 03:25:00 host1 CentralProcess1[9386]: DEBUG,HTTP Session received
notification,3021577136,65538,HTTPSession::ReceiveDataCallback,"handler_id=1, session_id=1212"
EventID=46, LogLine=Sep 20 03:25:11 host1 CentralProcess1[9386]: VERBOSE,SMTP new
job,3048000432,65549,SMTPClientHandler::Handle_Rcpt,"handler_id=13,socket=172,mail_from=test@test.co
m,rcpt_to=test@test.com"
```

Figure 3.18: Actual log lines represented by event 30 and event 46

Further investigation shows that each group of these similar patterns actually represents a system behavior and the difference between the log patterns in the same group describes the different code paths or states. To help readers understand it better, we draw the following execution paths for the log event 30 and the log event 46.

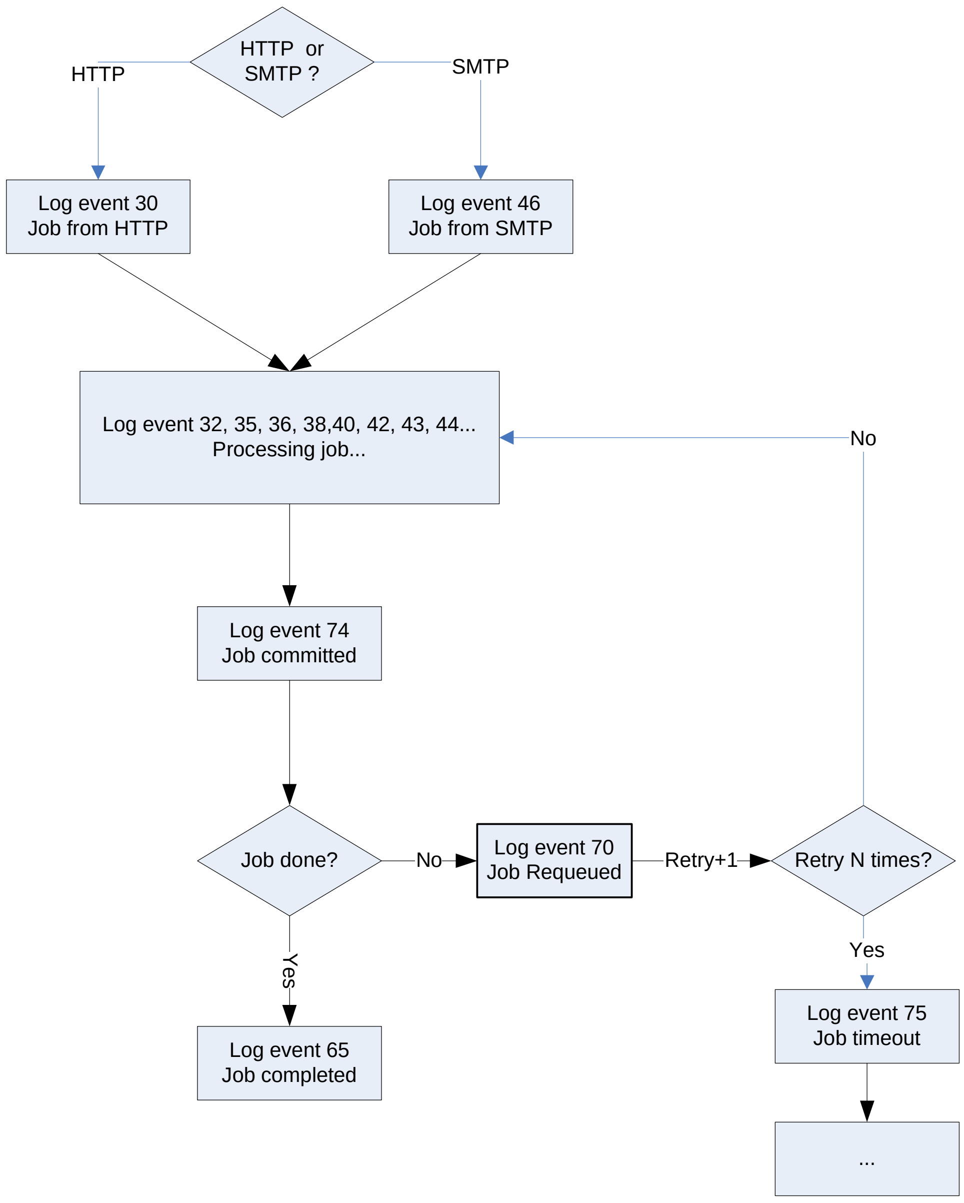

HTTP or SMTP ?
HTTP
SMTP
Log event 30 Job from HTTP
Log event 46 Job from SMTP
Log event 32, 35, 36, 38,40, 42, 43, 44... Processing job...
No
Log event 74 Job committed
Job done?
No
Log event 70 Job Requeued
Retry+1
Retry N times?
Yes
Yes
Log event 65 Job completed
Log event 75 Job timeout
...

Figure 3.19: Code paths for the log events 30 and 46

As we can see in Figure 3.19, t. The only difference is that the messages represented by the log pattern 1 come from the HTTP mail server and the messages identified as log pattern 2 come from the SMTP server.

The major reason why we need to group these patterns is that by grouping the similar log patterns, we would be able to understand what system behaviors can be observed in the log data. Furthermore, if we use these groups of log patterns as input data for the chi-square goodness of fit test, we cannot only determine if there is a problem in the system but also identify which groups of system behaviors have been changed. The abnormal groups can be very useful information when we try to detect the root cause of the problems.

The grouping algorithm is very similar to the one used for the log event clustering. We calculate the similarities between two patterns by dividing the number of identical event IDs of two patterns by the smaller number of the list of the log event IDs. For instance, if two patterns are both composed of 10 events and nine out of the events are identical, then the similarity can be calculated as 9 divided by 10, which is 90%. A more concrete example is shown in the follow equation by computing the similarities between the log pattern 1 and log pattern 2 in Figure 3.17.

$$\textit{Similarity\_Of\_Pattern\_1\_and\_Pattern\_2} = 18/19 = 94.7\%$$

Equation 3.3: Formula to compute similarities between two log patterns

Since the result 94.7% is higher than the default value of the similarity threshold (80%), log pattern 1 and log pattern 2 should be grouped together. The final grouping results of the log patterns in Figure 3.17 are shown in the following figure.

```
GroupID=1, PatternID=4, Events=30,31,32,35,36,37,38,40,42,43,44,48,54,55,56,57,58,59,60,61,62,63,65,74,
GroupID=1, PatternID=5, Events=31,32,35,36,37,38,40,42,43,44,46,48,54,55,56,57,58,59,60,61,62,63,65,74,
GroupID=1, PatternID=9, Events=31,32,35,36,37,38,40,42,43,44,46,54,55,56,57,58,59,60,61,62,63,74,75,76,77,
GroupID=1, PatternID=10,
Events=30,31,32,35,36,37,38,40,42,43,44,54,55,56,57,58,59,60,61,62,63,74,75,76,77,
GroupID=1, PatternID=11, Events=30,31,32,35,36,37,38,40,42,43,44,54,55,56,57,58,59,60,61,62,63,65,70,74,
76,77
GroupID=1, PatternID=12, Events=31,32,35,36,37,38,40,42,43,44,46,54,55,56,57,58,59,60,61,62,63,65,70,74,
76,77

GroupID=2, PatternID=6, Events=31,32,42,43,46,47,48,54,55,56,57,62,63,65,74,
GroupID=2, PatternID=7, Events=30,31,32,42,43,47,54,55,56,57,62,63,65,74,
GroupID=2, PatternID=13, Events=31,32,42,43,46,47,54,55,56,57,62,63,65,70,74,76,77,
```

Figure 3.20: Results of grouped log patterns

The format of the grouping result can be described as below:

GroupID=m, PatternID=n, Events=$LIST_OF_LOG_EVENTS

The GroupID is a unique integer that identifies a group of log patterns. The PatternID field represents the unique identifier of a log pattern and the LIST_OF_EVENTS_ASC field is composed of a sequence of log events that could be correlated by the same message ID. The list of log event ID is sorted in ascending order.

3.8 Parameters for Chi-square Test

In this section, we will discuss some important parameters used in our chi-square test.

3.8.1 Time Window

Before we run the chi-square test, we need to perform frequency analysis over the data of the log patterns to generate input data. Therefore, the first parameter in our test is the length of a time window where log data will be aggregated. For instance, if we define the time window length as 1 minute, then each row in the following table represents the statistical information of log patterns during a one minute period.

Group1	Pattern_2	Pattern_4	Pattern_7	Pattern_8	…
20081031153644.00	6	4	7	0	…
20081031154644.00	4	6	5	0	…
20081031155644.00	5	0	5	0	…
20081031160644.00	7	0	14	0	…
20081031161644.00	3	0	4	0	…
…	…	…	…	…	…

Table 3.6: Examples of log pattern data with one minute time window

3.8.2 Baseline

In this thesis, a baseline is defined as a sequential time windows where the system activities are considered normal. We compare the data in a test time window against the data in the baseline to determine if the distribution of log patterns in the new windows fits the distributions in the baseline. The total length of the baseline equals to the size of the baseline times the length of a time window. Considering the time window to be one minute in Table

3.6, if we set the size of baseline window to five, then the log data in the first five minutes will be regarded as the first baseline.

3.8.3 Our Sliding Window Approach

As the behaviors in an Internet service can vary in different periods, it is not practical to use the data in a fixed baseline to represent the normal activities of the whole day. To resolve this problem, we need to introduce a sliding window technique that allows the baseline to shift. Therefore, we develop an algorithm that controls how to shift or freeze the sliding window. The sliding window algorithm works as follows:

First, we choose the first B time windows in the log data as the first baseline and the B+1 time window as the current window. We then compare the distribution in the current time window against the first baseline. If the chi-square test result rejects the null hypothesis that the data is normally distributed if compared to the baseline, we believe that there are potential faults in the current time window. Here we refer fault-free samples as "Good" windows and faulty samples as "Bad" windows. We temporarily mark the current time window as a "Bad" window, as illustrated in Figure 3.21, and continue testing the data in the rest of the N-1 time windows.

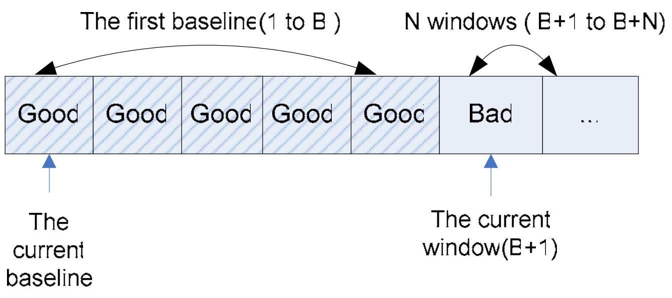


Figure 3.21: An example of a sliding window (B=5, N=2)

If the test result shows that the time windows from B+1 to B+N are not all "Bad" windows, we consider the N+1 window previously marked as a "Bad" window to be a potential false alarm and hence change the status of the N+1 window from "Bad" to "Good". Then we slide the baseline by one window and move the current time window to B+2, as shown in Figure 3.22. Now we will compare the data in the B+2 window against the second baseline line composed of windows from 2 to B+1.

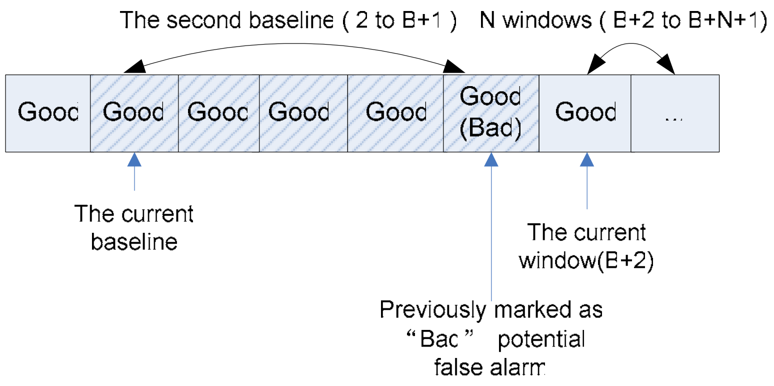

The second baseline (2 to B+1)
N windows (B+2 to B+N+1)
Gooc
Good
Gooc
Good
Good
Good (Bad)
Good
...
The current baseline
The current window(B+2)
Previously marked as "Bac" potential false alarm

Figure 3.22: A potential false alarm (B=5, N=2)

On the contrary, if the chi-square test results indicate that all the next N time windows are "Bad" windows, as shown in Figure 3.23, we believe that a problem actually occurred within the B+1 time window. Therefore, we formally mark the B+1 window as "Bad" and then set the system status to abnormal.

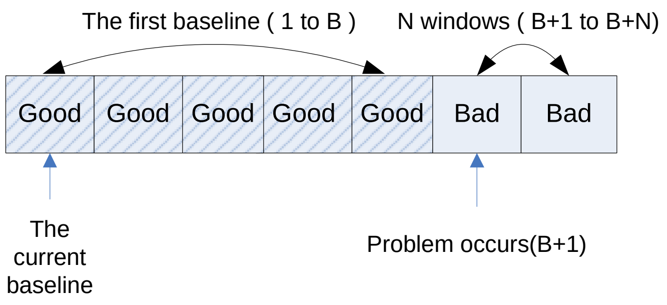


Figure 3.23: A problem occurs (B=5, N=2)

Once we detect a problem, we freeze the baseline instead of sliding to the next window. As shown in Figure 3.24, we move the test window to the B+2 window and compare the B+2 time window against the frozen baseline.

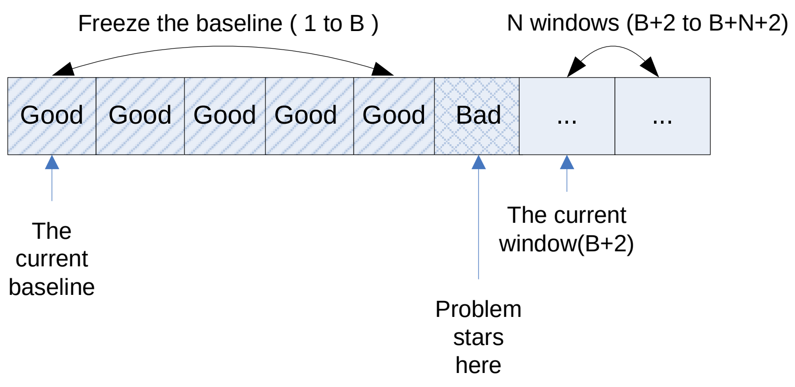


Figure 3.24: Freeze the baseline (B=5, N=2)

Furthermore, if we find that the X to X+N time windows are all "Good" windows and the current system status is abnormal, we believe the system has recovered somewhere during the time window X. Therefore, we slide the beginning of the baseline from 1 to X and also move the current test time window to X + B. The time span that the problem lasts should be X–B windows.

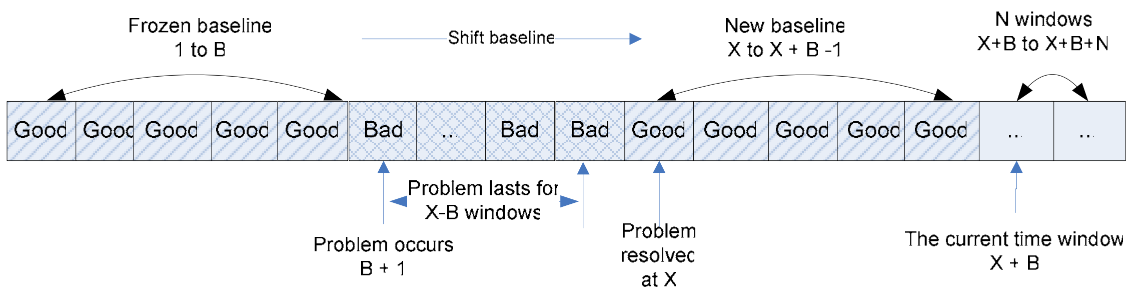


Figure 3.25: Baseline is shifted after the problem is resolved (B=5, N=2)

The above approach will be applied to all time windows in the data set, so eventually we could generate a history diagram showing the status changes of the system.

```
T=total number of time windows
B=baseline size
N=the number of continuous “Bad” or “Good” windows to be considered as a change in the system
TimeWindow=array of information about the system
CurrentBaselinePosition = B
CurrentSystemStatus=”normal”
for i = B … T-N do
    if CurrentSystemStatus is “normal” then
        if TimeWindow[i] is temporarily detected as a “BAD” window then
            FalseAlarm ← false
            for j = i+1 … i+1+N do
                if TimeWindow[j] is not a “BAD” window then
                    FalseAlarm ← true
                    break
                end if
            end for
            if FalseAlarm is false then
                TimeWindow[i] ← “Good”
                Shift the baseline by 1
                CurrentBaselinePosition ← CurrentBaselinePosition + 1;
            else
                CurrentSystemStatus ← “abnormal”
                TimeWindow[i] ← “Bad”
                Freeze the baseline
            end if
        end if
    else
        SystemRecovered←true;
        for j = i … i+N do
            if TimeWindow[j] is detected as a “BAD” window then
                SystemRecovered ← false;
                break
            end if
        end for
        if SystemRecovered is true then
            CurrentSystemStatus ← “normal”
            TimeWindow[i] ← “Good”
            Shift the baseline and current time window by B
            CurrentBaselinePosition ← CurrentBaselinePosition + B;
            i ← i + B
        end if
    end if
end for
```

Figure 3.26: The sliding window algorithm

3.9 Input for Chi-square Test

In our framework, we are interested in two types of metrics. The first type is the numbers of log patterns observed in each time window. Our expectation is that whenever a problem occurs, the developers should log warning and error messages. Thus, the log patterns representing abnormal behaviors will be generated and a different distribution against the baseline is created. The second metric is the processing time of messages of different patterns. We expect the performance issues in an Internet service can be detected based on this metric.

3.9.1 Metric One: The Numbers of Log Patterns

If we assume the length of time window is set to one minute, then each row in Table 3.7 represents the number of different patterns observed in one minute. The timestamp column indicates the beginning of that particular time window.

Time stamp	Pattern_2	Pattern_3	Pattern_4	Pattern_5	Pattern_7	Pattern_8	Pattern_13	Pattern_14
20081031153644	6	7	4	0	7	0	3	0
20081031154644	4	42	6	0	5	0	3	0
20081031155644	5	6	0	0	5	0	0	2
20081031160644	7	8	0	0	14	0	3	0
20081031161644	3	5	0	0	4	0	1	0
…	…	…	…	…	…	…	…	…

Table 3.7: Examples of frequency data for log patterns

In section 3.7, we have discussed the similarities among log patterns. Before performing the chi-square test, we divide the original frequency table into multiple tables so that each table contains only the information of the log patterns in the same group. The chi-square test will be performed based on the frequency data of each group instead of all groups. For instance, the frequency data in Table 3.7 is split into two groups in Table 3.8 and Table 3.9.

Group 1	Pattern_2	Pattern_4	Pattern_7	Pattern_8	…
20081031153644	6	4	7	0	…
20081031154644	4	6	5	0	…
20081031155644	5	0	5	0	…
20081031160644	7	0	14	0	…
20081031161644	3	0	4	0	…
…	…	…	…	…	…

Table 3.8: Examples of frequency data based on the number of log patterns in group 1

Group 2	Pattern_3	Pattern_5	Pattern_13	Pattern_14	…
20081031153644	7	0	3	0	…
20081031154644	42	0	3	0	…
20081031155644	6	0	0	2	…
20081031160644	8	0	3	0	…
20081031161644	5	0	1	0	…
…	…	…	…	…	…

Table 3.9: Examples of frequency data based on the number of log patterns in group 2

3.9.2 Metric Two: Processing Time

The way we calculate the processing time is to find the interval between the timestamp of the first log line and the timestamp of the last log line that can be correlated by the same message ID.

```
The first log line: Sep 20 03:25:00 host1 CentralProcess1[9386]: DEBUG,HTTP Session received
notification,3021577136,65538,HTTPSession::ReceiveDataCallback,"handler_id=1, session_id=1212
The last log line: Sep 20 03:25:01 host1 CentralProcess1[9386]: DEBUG,get user
result,47610800,65538,UserManager::GetUser_Result," get user
result,asyncid=65538,user_id=123456789,srp_id=TEST,time=15 msec
```

Figure 3.27: The first and the last log line related to the same message

For instance, as shown in Figure 3.27, the processing time for the message with message ID 65538 is 1 second, which is calculated by subtracting 03:25:01 by 03:25:00. Since the log patterns of the message have been already identified at this point, we can calculate the average processing time for each log pattern by running some complex SQL queries to the embedded database SQLite. First, we select the number of messages identified as a particular pattern in a time window using SQL query 1 in Figure 3.28. Then we compute the processing time for each individual message using SQL query 2. The result min_timetamp is defined as the timestamp of the first log line of the multi-lines for a message and the result max_timestamp refers to the timestamp of the last log line.

```
SQL query 1: select log_pattern_id, count(distinct(message_id)) from la_log_line  where log_pattern_id=%d
and message_id != 0 and time_stamp>%d and time_stamp<%d

SQL2 query2 :select message_id, min(time_stamp) as min_time, max(time_stamp) as max_time from
la_log_line where time_stamp>=%d and time_stamp<%d and message_id in (select distinct(message_id) as
distinct_message_id from la_log_line where log_pattern_id=%d and time_stamp>=%d and time_stamp<%d)
group by message_id
```

Figure 3.28: Examples of SQL queries to the embedded SQLite database

The processing time of a message can be calculated using the following equation.

Processing time = max_timestamp – min_timestamp

3.9.3 Final Input: Merge the Two Metrics

The next step is to divide the messages into two bins. We define a parameter called BIN_INTERVAL so that the messages with processing time less than or equal to BIN_INTERVAL will be placed into bin 1 while the messages with processing time more than BIN_INTERVAL will be placed into bin 2. Let us assume every message should be completed in 1 second in the system according to our performance requirement. Therefore, messages will be put to two bins depending on whether the message is completed in 1 second or not. The original data in Table 3.10 will be split into two bins demonstrated in Table 3.11.

Row num	Timestamp	Pattern 1	Pattern 2
1	20090204165026	102	43
2	20090204165126	97	38
3	20090204165226	101	41
4	20090204165326	103	42
5	20090204165426	102	41
6	20090204165526	96	34
7	20090204165626	**93**	39
8	20090204165726	38	15
9	20090204165826	45	18
10	20090204165926	49	21

Table 3.10: Examples of number of patterns before being divided into two bins

Row num	Timestamp	Pattern1_bin1	Pattern1_bin2	Pattern2_bin1	Patter2_bin2
1	20090204165026	100	2	42	1
2	20090204165126	96	1	38	0
3	20090204165226	101	0	41	0
4	20090204165326	103	0	42	0
5	20090204165426	102	0	41	0
6	20090204165526	94	2	33	1
7	20090204165626	**87**	**6**	36	3
8	20090204165726	17	21	6	9
9	20090204165826	26	19	11	7
10	20090204165926	33	16	15	6

Table 3.11: Examples of number of patterns after being divided into two bins

The columns Pattern1_bin1 and Pattern1_bin2 represent the two bins for pattern 1. The number in each cell indicates the number of messages where the processing time falls into that particular bin during that time window. For example, there are total 93 messages identified as pattern 1 in the seventh time window, which is shown in the seventh row in Table 3.10. Since the processing time of 87 messages is less than 1 second while the remaining six messages have a processing time of more than 1 second, the original 93 messages are divided into two bins: Pattern1_bin1 with 87 messages and Pattern1_bin2 with 6 messages, as shown in the 7th row in Table 3.11.

3.10 Chi-square Goodness of Fit Test

In statistics, a chi-square goodness of fit test describes how well a statistical model fits a set of observations [19]. In this thesis, the purpose of the chi-square test is to detect if the observed system behaviors in the new log data fit the statistical model built based on history log data in the baseline.

The chi-square statistic is a sum of differences between observed and expected outcome frequencies, each squared and divided by the expectation:

$$X^2 = \sum \frac{(O-E)^2}{E}$$

Equation 3.4: Chi-square function

Where:

O = an observed frequency

E = an expected (theoretical) frequency, asserted by the null hypothesis

The result of chi-square can be compared to the chi-square distribution table to determine the goodness of fit.

3.10.1 Degrees of Freedom

In statistics, the phrase degree of freedom is used to describe the number of values in the final calculation of a statistic that are free to vary [17]. In order to determine the degrees of freedom in a chi-square distribution, one takes the total number of columns of observed frequencies, subtracts one and then multiples by the number of rows of observed frequencies minus one.

$$Degree\_Of\_Freedom = (Number\_Of\_Rows - 1) * (Number\_Of\_Columns - 1)$$

Equation 3.5: The formula to calculate degree of freedom

In our tests, the maximum number of patterns found in the log data sets could be more than 300, which results in a very large degree of freedom. After grouping log patterns using the algorithm in the previous section, the number of features is significantly reduced in the chi-square test performed within each group.

3.10.2 Chi-square Distribution Table

Table 3.12 shows the critical values of the chi-square distribution with the corresponding probability and degrees of freedom. To determine the critical value of a chi-square distribution with a specific degree of freedom, we could go to the given probability column and the desired degree of freedom row. For example, the critical value for a chi-square test with .05 probability and four degrees of freedom is 9.48773, as shown in the italic and bold text in Table 3.12.

Probability / Degree of Freedom	.250	.100	.050	.025
1	1.32330	2.70554	3.84146	5.02389
2	2.77259	4.60517	5.99146	7.37776
3	4.10834	6.25139	7.81473	9.34840
4	5.38527	7.77944	***9.48773***	11.14329
5	6.62568	9.23636	11.07050	12.83250
6	7.84080	10.64464	12.59159	14.44938
7	9.03715	12.01704	14.06714	16.01276
8	10.21885	13.36157	15.50731	17.53455
…	…	…	…	…

Table 3.12: Critical values of chi-square distribution

Since a chi-square probability of less than or equal to 0.05 (or the chi-square statistic being at or larger than the 0.05 critical point) is commonly used in chi-square tests[18], we also choose 0.05 as the default probability for our chi-square test. If the chi-square value computed in our test is equal to or higher than the critical value given a 0.05 probability, then we should reject the null hypothesis that the data are normally distributed. In other words, the observed behaviors in the new log data do not fit the statistical model built based on the log data in the baseline. The most possible reason would be that there is a significant change in the system behaviors.

The chi-square test can be used to assess two types of comparisons: tests of goodness of fit and tests of independence. In our thesis, we use the first type of the chi-square distribution to determine whether the test data fits the observed distribution in the baseline. For instance, if we assume each row represents the data in a time window and the size of sliding window is three, then the first three rows of data will be regarded as the baseline that represents normal behaviors in the system. The purpose of the first chi-square goodness of fit test is to find out whether the data in the current time window, the 4th row, follows the distribution of the log patterns observed in the first three rows.

	Column 1	Column 2	Column 3	Row total
Row 1	a	b	c	R1=a+b+c
Row 2	d	e	f	R2=d+e+f
Row 3	g	h	i	R3=g+h+i
Column total	C1=a+d+g	C2=b+e+h	C3=c+f+i	T=a+b+c+d+e+f+g+h+i
Row 4	j	k	l	R4=j+k+l

Table 3.13: How to calculate chi-square for goodness-of-fit Test

Since the expected values of the fourth row are still unknown at this time, we need to calculate them based on the first three rows. In general, the "theoretical frequency" for any cell (under the null hypothesis) in a goodness of fit test is calculated as [18]:

$$E_i = N / n$$

For instance, the expected value of cell [4, 1] (row 4, column 1) should be

$$E_{[3,1]} = (a + d + g) / 3$$

However, such an expected value does not fit the dynamic workload in our log training data. Considering the log data collected at midnight when most people are not working, there would be less email traffic in the system. For instance, the amount of email traffic would gradually decrease to a small amount at midnight, as shown in Table 3.14.

Time span	Number of log pattern1	Number of log pattern2	Number of log pattern3

10:00pm to 11:00pm	2000	200	20
11:00pm to 12:00pm	1000	100	10
12:00pm to 1:00am	500	50	5
1:00am to 2:00am	250	25	2

Table 3.14: Network traffic around the midnight

If we use the average mean as expected value, then the expected value for cell [4, 1] (1:00 am to 2:00 am) would be (2000 + 1000+ 500)/3 = 1167. Hence the computed chi-square value will be much higher than the critical value, which means the distribution of the current time window (the 4th row) does not fit the distribution in the current baseline (the 1st to the 3rd rows). However, in reality, if we look at the big picture, the numbers of all log patterns decrease at the same pace, which perfectly matches the nature trend of workload in an Internet service like BIS during midnight. In such a case, we could not simply draw a conclusion that there is a significant change or a problem in the system.

To resolve this issue, we develop use a modified version of the method originally used to calculate the expected values in chi-square test of independence. This method shows a very good result in our test. In the original chi-square independence test, the expected value of cell [4, 1] (row 4, column 1) should be calculated as below

$$\text{Expected Value[4,1]=}\left(\frac{(a+d+g+j)}{a+b+c+d+e+f+g+h+i+j+k+l}\right)*(j+k+l)=\left(\frac{C1+j}{T+R4}\right)*R4$$

In our modified version, since the last row should not be included in the baseline, the formula to compute the expected value [4, 1] is changed as below.

$$\text{Expected Value [4, 1] =}\left(\frac{(a+d+g)}{a+b+c+d+e+f+g+h+i}\right)*(j+k+l)=\left(\frac{C1}{T}\right)*R4$$

So the corresponding chi-square value of cell [4, 1] can be calculated as

$$X_{[4,1]}{}^{2}=\frac{\left(g-\left(\frac{C1}{T}\right)*R4\right)^{2}}{\left(\frac{C1}{T}\right)*R4}$$

And the final chi-square value should be $X_{4,c}{}^{2}=\sum_{c=1}^{3}\frac{(O_{[4,c]}-E_{[4,c]})^{2}}{E_{[4,c]}}$

We believe such a formula is more suitable to our log analysis because it reduces the false alarms introduced by the reasonable changes of system behaviors or network traffic in a message-flow Internet service. The biggest difference between our method and the chi-square test of independence is that we remove the last row from the total value. On the contrary, the last row will be included in the total value in the chi-square test of independence.

In a chi-square goodness of fit test, the degree of freedom equals to the number of log patterns minus one. Entering the chi-square distribution in Table 3.12 with a given degree of freedom and a 0.05 probability, we can find the corresponding critical value.

If the chi-square value of the current time window is smaller than the computed critical value, we accept the null hypothesis that the new row and the sliding window have equal or similar distribution. In other words, we believe there are no significant changes of log patterns in the current time window when compared to the baseline.

Once the chi-square test is done in the current time window, we shift or freeze the baseline and the current time window according to the test result. The detailed steps and related algorithm has already been discussed in the section 3.8.3.

Chapter 4

Evaluation

In this chapter, we discuss seven log data sets generated by the Blackberry Internet Service (BIS) system to evaluate our solution. We inject faults in a simulation environment and evaluate how well our proposed approach fares in detecting anomalies.

4.1 Simulation Environment

Our simulation test-bed, shown in Figure 4.1, includes a workload generator called EmailSimulator, two components including CentralProcess1 and CentralProcess2 that handle most of the business logic, a database application server that stores user information, and a GSM Network Simulator that simulates the requests and responses from Blackberry devices. There are two major types of messages in the simulation environment. The first one is called Message To Handheld (MTH), which is sent from the EmailSimulator to the BIS engine and then delivered to the handheld. The other one is called Message From Handheld (MFH) that is sent from handheld to BIS and eventually delivered to the EmailSimulator.

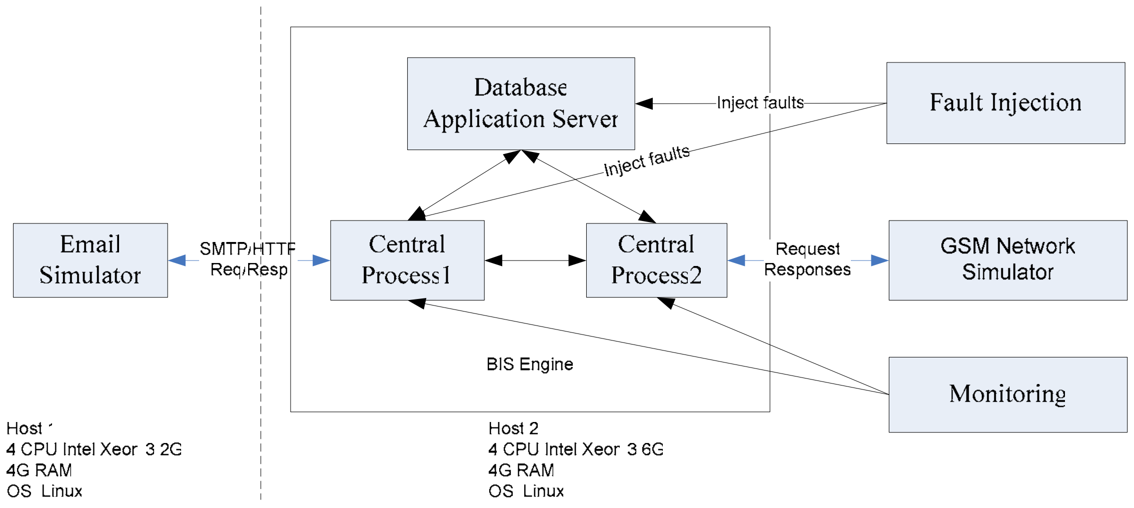


Figure 4.1: Architecture of the simulation environment

The EmailSimulator generates steady MTH traffic by sending a fixed number of emails per second to the CentralProcess1 server via SMTP and HTTP ports. The MTH messages travel through the CentralProcess1 server, the database application server, and the CentralProcess2 server. Eventually it arrives at the GSM Network Simulator. At the same time, the GSM Network Simulator generates MFH messages that will be delivered to the EmailSimulator in a reversed order from which the MTH travels.

In our test, we send two MTH messages (one via SMTP and one via HTTP) per second from the EmailSimulator to the CentralProcess1, as well as one MFH message per second from the GSM Network Simulator to the CentralProcess2. We also develop a fault injection module that uses some scripts to inject faults into the systems to simulate different types of problems. The monitor module collects the corresponding log files and system statuses from the simulation environment.

4.2 Parameters for the Simulation Environment

Parameter	Value	Description
Time window length	1 minute	The number of log patterns in the log files will be aggregated for each period of 1 minute.
Baseline size	5	A baseline is composed of 5 continuous time windows
N support for sliding window	2	A problem is detected if there is a sequence of N windows where the distribution of the observed log patterns does not fit the baseline.
Normal processing time	1 second	Data is divided into 2 bins: Messages completed in 1 second will be placed in BIN1; otherwise they will be placed in BIN2
BIN number	2	

Table 4.1: Parameters for chi-square test

The parameter “normal processing time” is based on our expectation for the system that all messages should be completed in 1 second in a LAN environment. As for other parameters, as long as there are enough samples generated during each time window, they should not have a significant impact on our test results. Moreover, the same parameters will be used to detect seven different types of faults, including network errors, application errors, and performance issues, so we can assume our problem detection algorithm does not heavily rely on the parameter tuning.

4.3 Case 1: Database Application Server Down

The first fault simulates a system error where a critical component such as the database server is down in the system. At the beginning, the simulation environment will be running without any problems for 25 minutes. Then we manually shutdown the database application server by running “kill -9 PID” commands on Linux. At this point, we expect the log patterns to change significantly because lots of warnings and errors should be logged when the database server is down. Since these warnings and errors would not be observed in the baseline, we have an expectation that the distribution of the log patterns during the downtime of the database does not fit the distribution in the baseline. After the database server has been shut down for 5 minutes, we restart the database server. In such a case, the log patterns are expected to change back to the ones in the baseline gradually. We repeat the shutdown-restart test scenario three times, so the entire test takes about 90 minutes ((25 + 5)*3) to run.

4.3.1 Log Data Set

Below is a summary table about the log data we collect in the first test case. There are total four log files generated by two components as each component writes logs to a log file that rotates every hour.

Name	Value
Number of log files	4
Number of components	2
Total log file size	70 Mbytes
Number of log lines generated by CentralProcess1	214,438
Number of log lines generated by CentralProcess2	67,885
Number of identified log events	61
Number of identified log patterns	12
Number of identified log pattern groups	4

Table 4.2: Log data information

```
GroupID=1, PatternID=1, Events=1,2,3,4,5,59,60,61,62,64,
GroupID=1, PatternID=10, Events=1,4,5,6,9,19,20,22,50,60,61,62,64,76,

GroupID=2, PatternID=2,
Events=1,4,5,6,9,19,20,22,33,35,38,39,40,41,43,44,50,60,61,62,63,64,65,70,71,72,73,82,83,87,
GroupID=2, PatternID=5,
Events=1,4,5,6,9,19,20,22,35,38,39,40,41,43,44,45,60,61,62,63,64,65,70,71,72,73,74,82,83,84,87,
GroupID=2, PatternID=6, Events=1,4,5,6,9,19,20,22,33,35,38,39,40,41,43,60,61,62,64,66,70,71,72,73,
GroupID=2, PatternID=11,
Events=1,4,5,6,9,19,20,22,35,38,39,40,41,43,44,45,51,52,53,60,61,62,63,64,65,70,72,73,74,82,83,84,87,
GroupID=2, PatternID=12,
Events=1,4,5,6,9,19,20,22,33,35,38,39,40,41,43,44,53,60,61,62,63,64,65,70,72,73,74,82,83,84,87,
GroupID=2, PatternID=13,
Events=1,4,5,6,9,19,20,22,35,38,39,40,41,43,44,45,50,60,61,62,63,64,65,70,71,72,73,82,83,87,

GroupID=3, PatternID=7, Events=33,35,38,39,40,46,47,64,66,69,73,77,85,
GroupID=3, PatternID=8, Events=33,35,38,39,40,46,64,66,69,71,73,77,85,

GroupID=4, PatternID=9, Events=35,38,39,40,41,46,64,67,70,72,73,85,86,
GroupID=4, PatternID=3,
Events=5,10,12,13,14,15,16,21,23,25,27,29,35,38,39,40,41,43,54,56,57,58,70,71,72,73,78,79,80,
GroupID=4, PatternID=4, Events=10,17,18,21,23,25,27,28,
```

Figure 4.2: Identified log pattern groups

There are four groups of log patterns identified in the log data. We manually review the log text and summarize each log pattern group in the following table.

Group ID	Description
1	Other Jobs
2	MTH (Message to Handheld)
3	Other Jobs
4	MFH (Message from Handheld)

Table 4.3: Descriptions of the log pattern groups

4.3.2 Chi-square Test Results

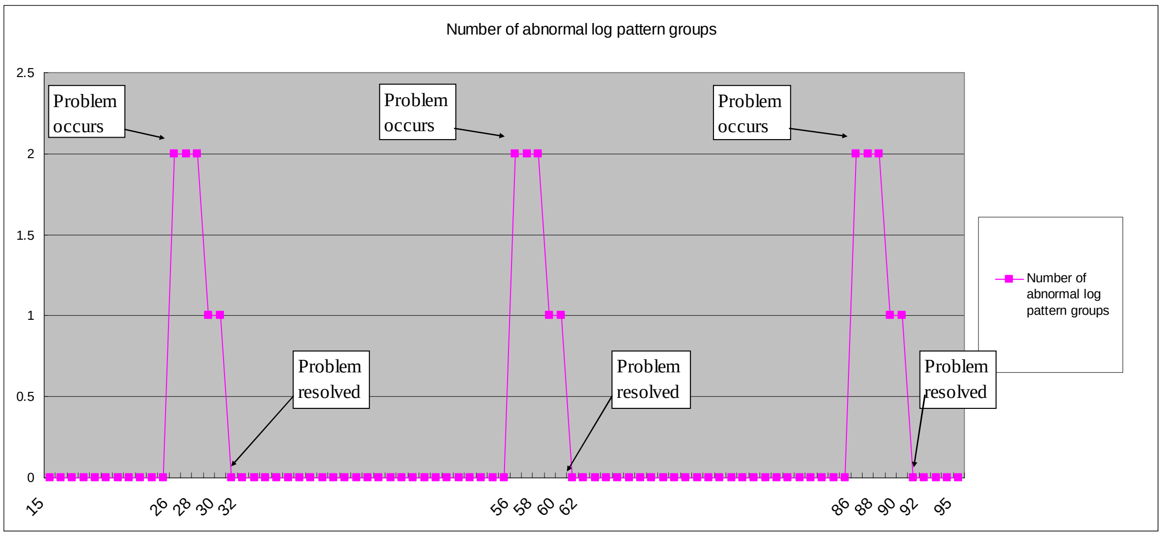


Figure 4.3: Number of abnormal log pattern groups

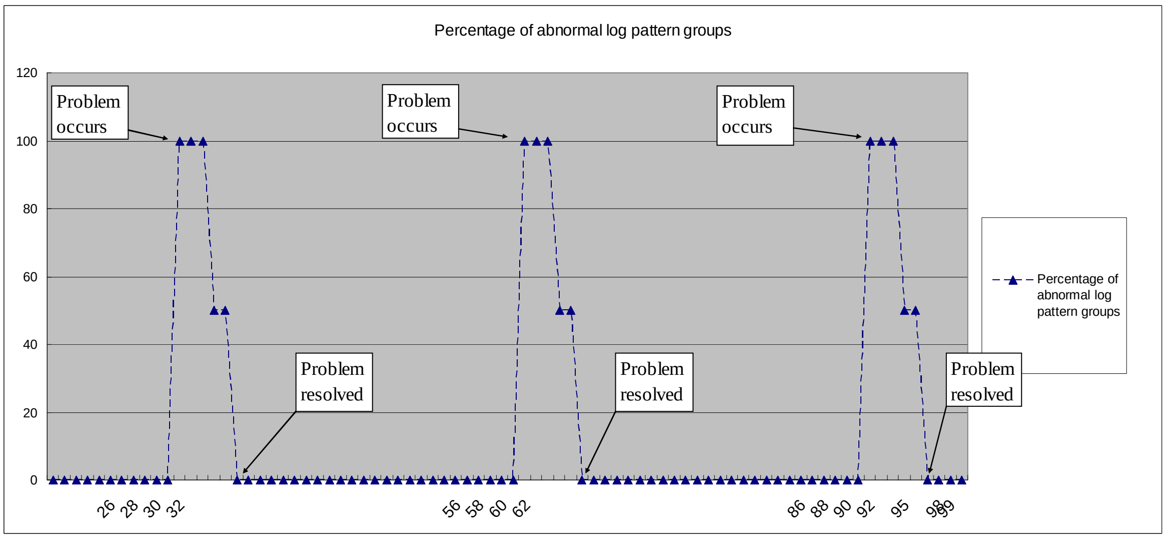


Figure 4.4: Percentage of abnormal log pattern groups

The solid line in Figure 4.3 represents the number of abnormal log pattern groups detected by the chi-square test using our sliding window approach discussed in 3.8.3. The dashed line in Figure 4.4 means the percentage of abnormal groups, which equals to the number of abnormal groups divided by the total number of groups observed during that particular time window. The boxes with text and arrows point to the time windows where the problem or the system recovery is detected by our framework.

Point	Time window	Descriptions of changes in the system
26	20091022213851 to 20091022213951	The first problem occurs at 20081022213806
32	20091022214351 to 20091022214451	The first problem is solved at 20081022214306
56	20091022220851 to 20091022220951	The second problem occurs at 20081022220806
62	20091022221351 to 20091022221451	The second problem is solved at 20081022221307
86	20091022223851 to 20091022223951	The third problem occurs 20081022223807
92	20091022224351 to 20091022224451	The third problem is solved at 20081022224307

Table 4.4: Associations between points and timestamp

As illustrated in Table 4.4, the faults and the recovery all occur in the middle of the time window and these six changes are all detected immediately in the next time windows. In the following sections, we will discuss the changes of system status one by one.

- Problem occurs: System status changes from normal to abnormal

Figure 4.3 shows that at the 26th, 56th and 86th points on the solid line, two groups of patterns are detected as abnormal groups in those time windows. At the same time, the dashed line also indicates 66.5% or 100% groups are marked as abnormal. These non-zero numbers indicate that a problem occurred in the system. The abnormal groups for this fault include group 2 and group 4, which are associated with MTH and MFH messages according to Table 4.3. This fault affects all components that depend on the database server. Therefore, we would suspect a shared component such as the base server is causing the problem.

Timestamp	Groups marked as abnormal	Number of log pattern 11
20091022220551.00	n/a	0
20091022220651.00	n/a	0
20091022220751.00	n/a	3
20091022220851.00	Group2; Group4	2
20091022220951.00	Group2; Group4	6
20091022221051.00	Group2; Group4	4
20091022221151.00	Group2	8
20091022221251.00	Group2	5

Table 4.5: Log pattern groups marked as abnormal when the first fault is injected

Furthermore, as shown in Table 4.5, there is a significant change in the number of pattern 11 in the MTH log pattern group when the problem occurs. As illustrated in Figure 4.5, this pattern actually represents the warning events that would be logged when a database server is not available. This finding confirms our suspicion that the root cause is in the database, so system administrators can take corresponding actions to resolve the problem.

```
…
LogEventID=51, LogLine=Oct 22 20:38:11 host1 CentralProcess1[21451]: WARNING,Connection lost
reconnect,31968176,67700,ConnectionPool::GrabConnection,"pool=LBAC_POOL,host=local”
LogEventID=52, LogLine=Oct 22 20:38:11 host1 CentralProcess1[21451]: WARNING,Set host not
available,31968176,67700,ConnectionPool::SetHostStatus,"pool=LBAC_POOL,host=local”
LogEventID=84, LogLine=Oct 22 20:38:11 host1 CentralProcess1[21451]:
[USER_CONNECTOR][DBConnectorManager::GetDatabaseServer],Event=grab_db_connection_failed,Desc
=asyncid=67538, jobid=67700, userid=123456789, serviceid=TEST, LOG_LEVEL=WARNING
…
```

Figure 4.5: Sample log events in the log pattern 11

Three minutes (three time windows) after the problem occurs, the number of abnormal log pattern group drops to one and the MFH group 4 previously marked as abnormal disappears on the figure. The reason is that the BIS system stops processing MFH messages when the database is down. In such a case, there would be not enough samples for the chi-square test to compare the distribution, so this MFH group is removed from the chi-square test result.

- Problems last: System status remains abnormal

The continuous non-zero values of abnormal groups in the five “Bad” windows indicate the problem lasts during the five minutes, which also matches the actual situation.

- Problems resolved: System status changes from abnormal to normal

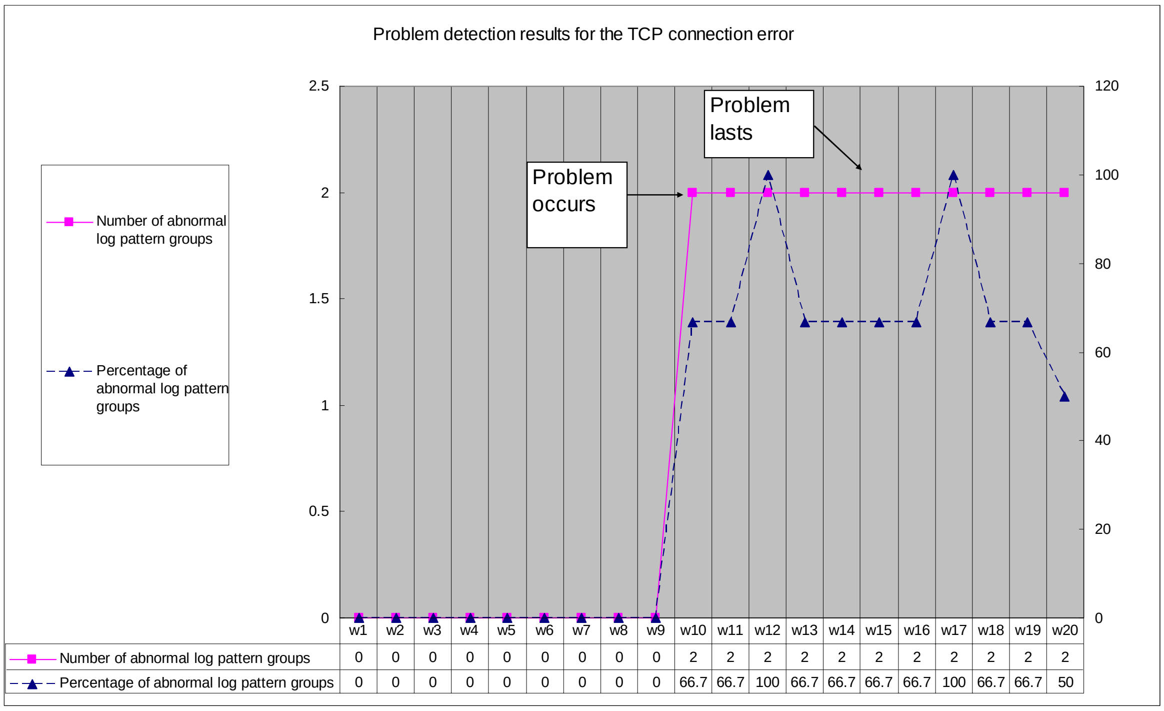
At the 32nd, 62nd and 92nd points, the values and the percentage of abnormal groups drops back to zero after the database server is restarted. This test proves that our algorithm is not only capable of detecting problems but also allows us to discover system recoveries, which can help us properly understand the status of the system.

4.4 Case 2: TCP Connection Error

This fault simulates a network error when the TCP connections from the CentralProcess1 and CentralProcess2 components are randomly closed by the database application server. We modify the source code of the database application server to explicitly close 25% of the TCP client connections by calling the standard close() socket API function. The whole experiment lasts 20 minutes. In the first 9 minutes, the system is running fault-free. At 9:01, we activate a fault in our target system, so the problem actually occurs in the 10th time window. According to our observation, once a TCP connection is closed, both of the CentralProcess1 and the CentralProcess2 components start to output warning and error messages in the event logs.

Name	Value
Number of log files	2
Number of components	2
Total log file size	33 Mbytes
Number of log lines generated by CentralProcess1	109,615
Number of log lines generated by CentralProcess2	28,917
Number of identified log events	71
Number of identify log patterns	12
Number of identified log pattern groups	4

Table 4.6: Log data information



	w1	w2	w3	w4	w5	w6	w7	w8	w9	w10	w11	w12	w13	w14	w15	w16	w17	w18	w19	w20
Number of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	2
Percentage of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	66.7	66.7	100	66.7	66.7	66.7	66.7	100	66.7	66.7	50

Figure 4.6: Problem determination results for the TCP connection error

Figure 4.6 shows that the problem is detected correctly and it lasts until the test is finished. Though this test result looks quite similar to the previous one, the major difference lies in how the messages are being retried in these two test cases.

```
GroupID=1, PatternID=1,
Events=1,2,5,6,7,17,18,20,30,31,32,34,35,36,37,40,42,43,44,63,64,65,66,67,68,69,70,71,72,85,86,
GroupID=1, PatternID=6,
Events=1,2,5,6,7,17,18,20,31,32,34,35,36,37,40,42,43,44,45,63,64,65,66,67,68,69,70,71,72,85,86,
GroupID=1, PatternID=7, Events=1,2,5,6,7,17,18,20,30,31,32,34,35,36,37,40,46,63,64,65,66,67,68,69,70,71,72,
GroupID=1, PatternID=8, Events=1,2,5,6,7,17,18,20,30,31,32,34,35,36,37,40,47,63,64,65,66,67,68,69,70,71,72,
GroupID=1, PatternID=13,
Events=1,2,5,6,7,17,18,20,31,32,34,35,36,37,40,45,46,63,64,65,66,67,68,69,70,71,72,
GroupID=2, PatternID=2,
Events=7,8,10,11,13,14,19,21,23,24,26,29,32,34,35,36,37,42,43,44,54,56,59,67,68,76,77,78,79,81,85,86,
GroupID=2, PatternID=4,
Events=7,8,10,11,13,14,19,21,23,25,26,29,32,34,35,36,37,49,54,56,59,62,67,68,73,76,77,80,82,84,85,86,87,90,
GroupID=2, PatternID=17, Events=7,8,10,11,13,14,19,21,23,26,29,32,34,35,36,37,49,54,59,60,68,77,85,86,89,
GroupID=3, PatternID=11, Events=30,31,32,34,35,36,37,49,50,65,68,70,73,85,86,
GroupID=3, PatternID=12, Events=31,32,34,35,36,37,45,49,50,52,53,65,68,70,73,85,86,
GroupID=4, PatternID=5, Events=8,15,16,19,26,28,
```

Figure 4.7: Identified log pattern groups

Group ID	Description
1	MTH (Message to Handheld)
2	MFH (Message from Handheld)
3	MTH(failures)
4	MFH (failures)

Table 4.7: Log pattern groups

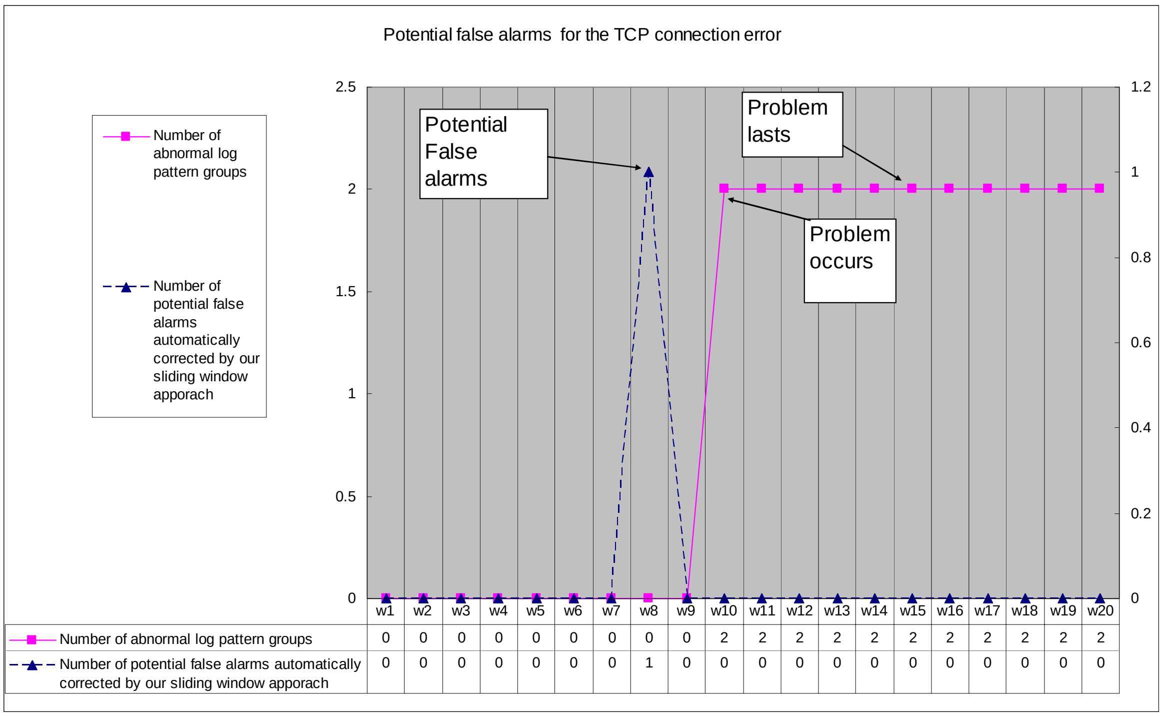
If we compare the log pattern groups between these two tests, we can find there is a big difference. In the previous test case, the CentralProcess1 and CentralProcess2 components keep trying to reconnect to the database server. However, they do not succeed because the database is down. The database is marked as unavailable and the whole system stops functioning. During the downtime, the whole system does not accept any new requests. Therefore, most of the MTH and MFH messages in the test case 1 are completed in 1 second when the database is up, which meets our expectation of the performance.

In this test, since we only randomly close 25% of the TCP connections. When a database client tries to reconnect, there is a 75% chance that a new TCP connection will be successfully established. Therefore, the entire system is still functioning. However, if the TCP connection is still not established after a certain numbers of retries, some messages would be considered as timed out and then dropped from the processing queue. Hence, the log pattern group 3 and log pattern group 4, which are two different code paths generated by the timeout warnings and error messages, can be observed during the “Bad” time windows. However, due to the low probability that a message will be actually dropped (e.g., if a message is tried 3 times, then the probability it will be dropped due to the TCP connection error is 25%*25%*25%=1.5625%), the number of samples of the abnormal groups generated during the “Bad” windows in fact does not meet the minimum number of samples required by a chi-square test.

Fortunately, each time when the same message is retried, the CentralProcess1 and CentralProcess2 will generate event logs that indicate the message with the same message ID is being resent. According to our method of calculating the processing time of a message, these retried messages will be put in BIN2 of each pattern. As illustrated in Table 4.8, the significant change between two BINS starting from 09:01 will be detected by our chi-square goodness of fit test.

Time window	Pattern1_bin1	Pattern1_bin2	Pattern6_bin1	Pattern6_bin2
6:01	196	1	136	2
7:01	200	3	141	3
8:01	201	0	141	0
9:01	94	22	46	11
10:01	89	29	46	13
11:01	78	35	35	15
12:01	78	26	28	17
13:01	80	25	27	17

Table 4.8: Number of log patterns in group 1 divided into two bins



	w1	w2	w3	w4	w5	w6	w7	w8	w9	w10	w11	w12	w13	w14	w15	w16	w17	w18	w19	w20
Number of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	2
Number of potential false alarms automatically corrected by our sliding window apporach	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.8: Potential false alarms automatically corrected for the TCP connection error

Figure 4.8 illustrates in time window 8, a potential false alarm is automatically corrected by our sliding window algorithm. As we discussed in Section 3.8.3, if and only if there are a sequential N (N=2 in this test) time windows identified as "Bad" windows, then a problem is confirmed. Otherwise, the current window temporarily marked as "Bad" window turns out to be a potential false alarm.

Time window	Timestamp	Groups temporarily marked as abnormal
W7	6:01	N/A
W8	7:01	Group1
W9	8:01	N/A

Table 4.9: Log pattern groups marked as abnormal in our chi-square test

As shown in Table 4.9, although the log pattern group 1 is temporarily marked as an abnormal group in the time window 8, it is not detected as a "Bad" window in the next time window 9. Hence, this potential false alarm is automatically corrected. The test result also proves that our sliding window technique is able to automatically detect and correct potential false alarms in our chi-square test.

In sum, our algorithm has been proven capable of detecting the following problems in test case 1 and test case 2:

Case 1: System errors represented by the log patterns of warning or error messages that are not observed in the baseline.

Case 2: If there are not enough samples of warning and error log patterns, we can still use the distribution of the two bins of messages to identify performance issues or abnormal behaviors caused by abnormal processing time.

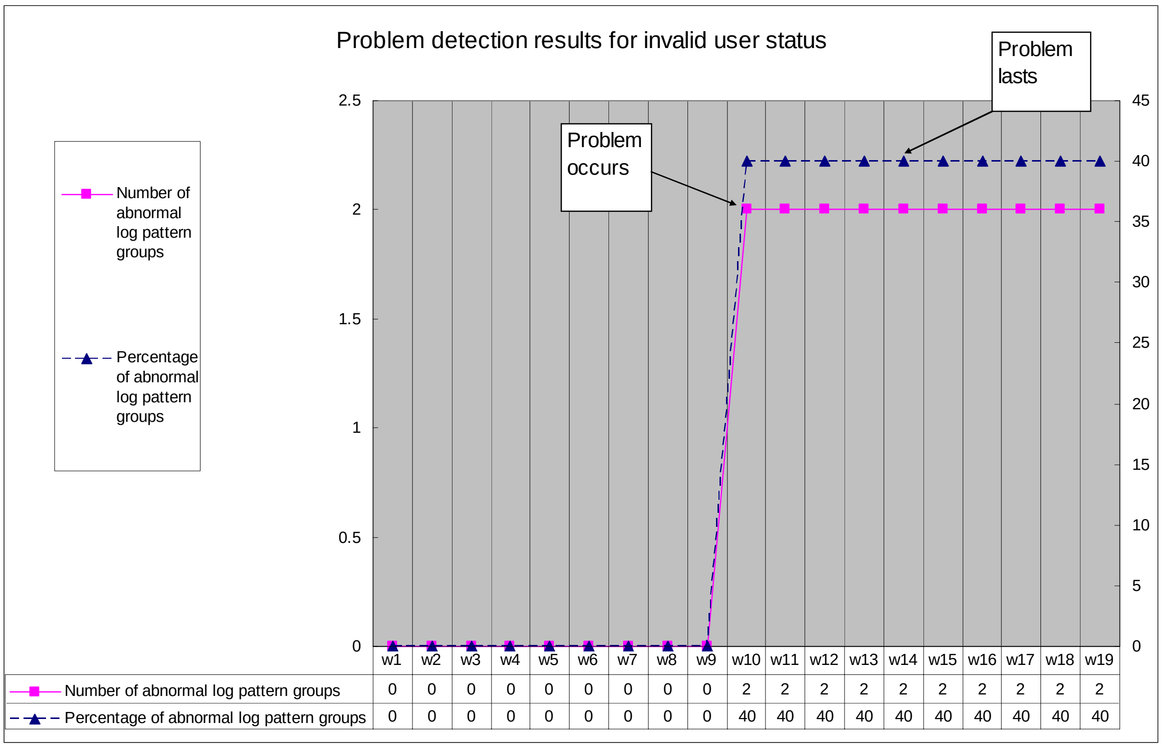
4.5 Case 3: Invalid User Status

In this test case, we simulate an application error where the statuses of 25% of users are accidentally marked as invalid in the database. This is a very common error when system administrators are performing database operations during system upgrades. The problem also occurs in the 10th time window, which is between 9:01 and 10:00. The Test result shows that the problem I correctly detected without any potential false alarms.

The log events and log patterns are similar to the ones in test case 2 except that the abnormal change in the log patterns is related to the user errors instead of TCP errors, so we are not going to give detailed analysis here.

Name	Value
Number of log files	2
Number of components	2
Total log file size	40 Mbytes
Number of log lines generated by CentralProcess1	144,294
Number of log lines generated by CentralProcess2	32,0000
Number of identified log events	76
Number of identify log patterns	16
Number of identified log pattern groups	5

Table 4.10: Log data information



	w1	w2	w3	w4	w5	w6	w7	w8	w9	w10	w11	w12	w13	w14	w15	w16	w17	w18	w19
Number of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2
Percentage of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	40	40	40	40	40	40	40	40	40	40

Figure 4.9: Problem detection results for the invalid user status

4.6 Case 4 to Case 7: Performance Issues

In the first three test cases, we have already simulated system errors such as unavailable hosts as well as application errors like invalid user statuses. In this section, we simulate performance issues in a system by manually injecting some delay into the processing of messages.

Since we assume most of the messages will be finished in less than a second in our testing environment, we define 2.5 seconds as a low-level delay and 5 seconds as a high-level delay. In like manner, 25% of user impact ratio is defined as a low-level impact and 50% of user impact ratio is defined as a high-level impact. All possible combinations of these two levels of problems are shown in the following table.

Test scenario	Low level delay(2.5 seconds)	High level delay(5 seconds)
Low-level impact ratio (25%)	Inject 2.5 seconds delay into 25% messages	Inject 5 seconds delay into 25% messages
High level impact ratio	Inject 2.5 seconds delay into 50% of	Inject 5 seconds delay into 50%

(50%)	messages	messages

Table 4.11: Combination of impact ratio levels and delay levels

For each of the impact levels and delay levels, the whole test lasts for 20 minutes. In the first 9 minutes, the system is running without any problems. At 9:01, we start to inject the fault, so the problem always occurs in the 10th time window. Afterwards, the problem lasts until the end of the test. We modify the source code in the database application server to call the sleep() function before sending back responses to the CentralProcess1 and CentralProcess2.

Name	25% ratio/2.5s delay	25% ratio/5.0s delay	50% ratio/2.5s delay	50% ratio/5.0s delay
Number of log files	4	2	2	4
Total log file size	22.6 M	20.3M	20.4M	18.9M
Number of log lines generated by CentralProcess1	81,121	71,686	72,177	65,023
Number of log lines generated by CentralProcess2	23,070	21,839	21,972	21,400
Number of identified Log events	51	59	68	61
Number of identified log patterns	7	8	5	9
Number of identified log pattern groups	2	2	2	3

Table 4.12: Log data information

The log data information about these four test cases is illustrated in Table 4.12. One interesting factor we can find in the table is that in most cases, if the impact ratio of two sets of log data is identical, then there would be more log events, more log patterns or more log groups identified in the log data set with higher levels of delay. For instance, the number of log events (59) and the number of log patterns (8) in the second column “25% ratio/5.0s delay” are larger than the values (51 & 7) of the corresponding cells in the first column. The same rule can be also observed in the data with identical delay but different impact ratio. Our explanation for such phenomenon is that when more users are get impacted by the problem or the injected delay is higher, the BIS engine system will naturally generate more types of warning and error logs and thus create more code paths. In the next sections, we describe the test result of each test scenario.

4.6.1 Case 4: 25% Impact Ratio and 2.5 Seconds Delay

```
GroupID=1, PatternID=1,
Events=1,2,5,6,7,14,15,17,18,19,20,22,23,24,25,26,29,30,31,32,35,36,40,48,49,50,51,52,53,54,55,56,
GroupID=1, PatternID=3,
Events=1,2,5,6,7,14,15,17,19,20,22,23,24,25,26,29,30,31,32,34,35,36,40,42,48,49,50,51,52,53,54,55,56,
GroupID=1, PatternID=4, Events=19,20,22,23,24,25,26,29,30,31,32,34,35,36,37,38,39,49,51,52,53,54,
GroupID=1, PatternID=5, Events=1,2,5,6,7,14,15,17,19,20,29,30,31,32,34,36,41,48,49,50,53,54,55,56,
GroupID=1, PatternID=6,
Events=1,2,5,6,7,14,15,17,18,19,20,22,23,24,25,26,29,30,31,32,42,48,49,50,51,52,53,54,55,56,
GroupID=1, PatternID=7, Events=1,2,5,6,7,14,15,17,18,19,20,29,30,31,32,41,48,49,50,53,54,55,56,

GroupID=2, PatternID=2,
Events=7,8,9,10,11,12,13,16,20,22,23,24,25,26,29,30,31,32,35,36,40,44,45,46,47,51,52,57,58,60,62,63,
```

Figure 4.10: Identified log pattern groups

Group ID	Description
1	MTH (Message to Handheld)
2	MFH (Message from Handheld)

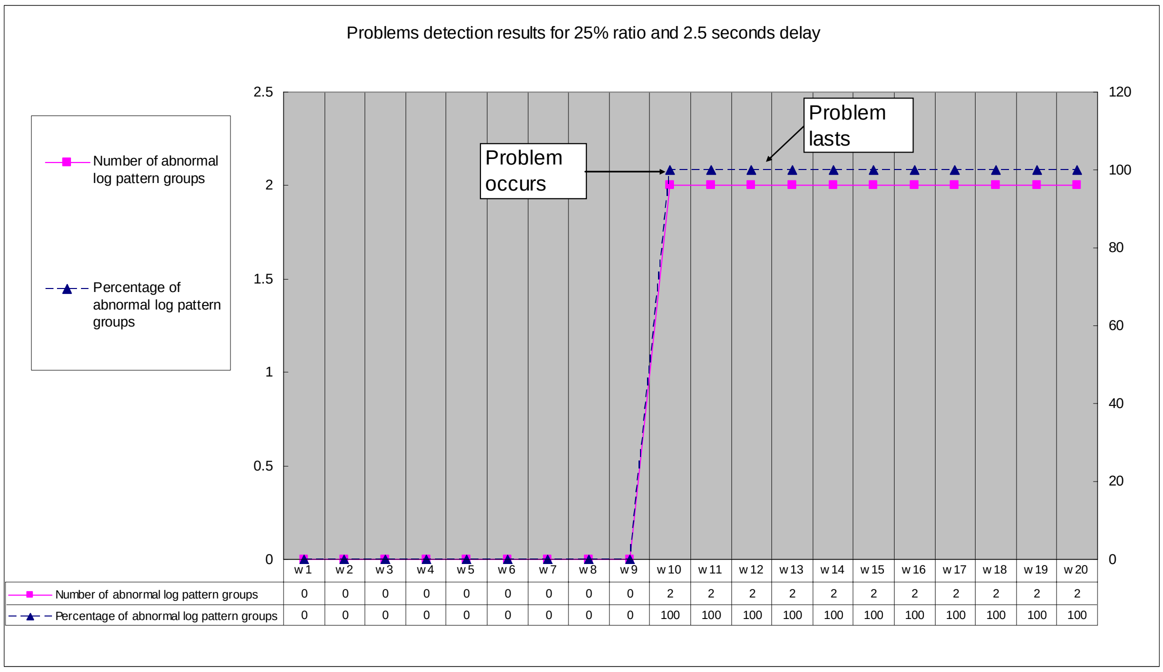
Table 4.13: Log pattern groups

There are only two groups of log patterns found in this case. In fact, after manually reviewing the events identified in this log data set, we could not find any warnings or errors. The reason could be that the database operation timeout is set to 3 seconds in our test, so no retried or timeout log events would be generated. Hence, the problem detected in this test case is purely based on the different distributions of message processing time.

Time window	Pattern1_bin1	Pattern1_bin2	Pattern6_bin1	Pattern6_bin2
6:01	105	1	43	1
7:01	104	0	42	0
8:01	95	3	34	1
9:01	82	12	32	7
10:01	4	36	2	14
11:01	8	39	2	16
12:01	12	37	6	15
13:01	13	33	2	11

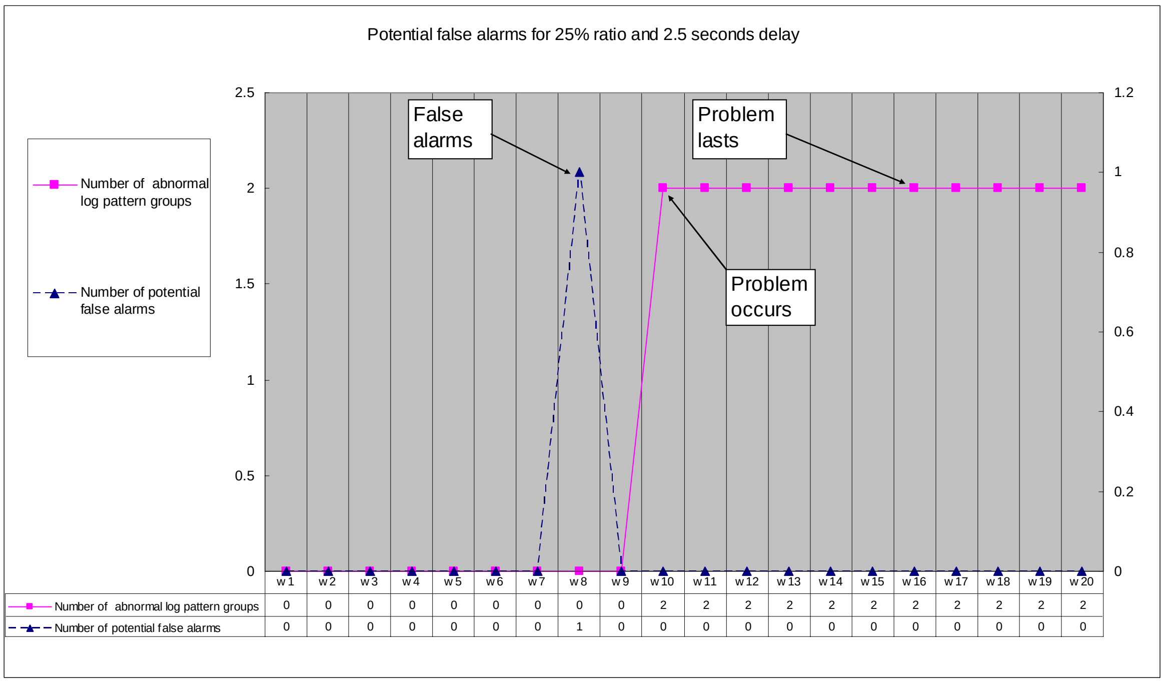
Table 4.14: Messages divided into bins based on processing time

As shown in Table 4.14, there is a significant change in the distribution between BIN1 and BIN2 starting from 9:01, which can be detected by our chi-square. The overall detection results are shown in the following figure.



	w 1	w 2	w 3	w 4	w 5	w 6	w 7	w 8	w 9	w 10	w 11	w 12	w 13	w 14	w 15	w 16	w 17	w 18	w 19	w 20
Number of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	2
Percentage of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	100	100	100	100	100	100	100	100	100	100	100

Figure 4.11: Problem detection results for 25% ratio and 2.5 seconds delay

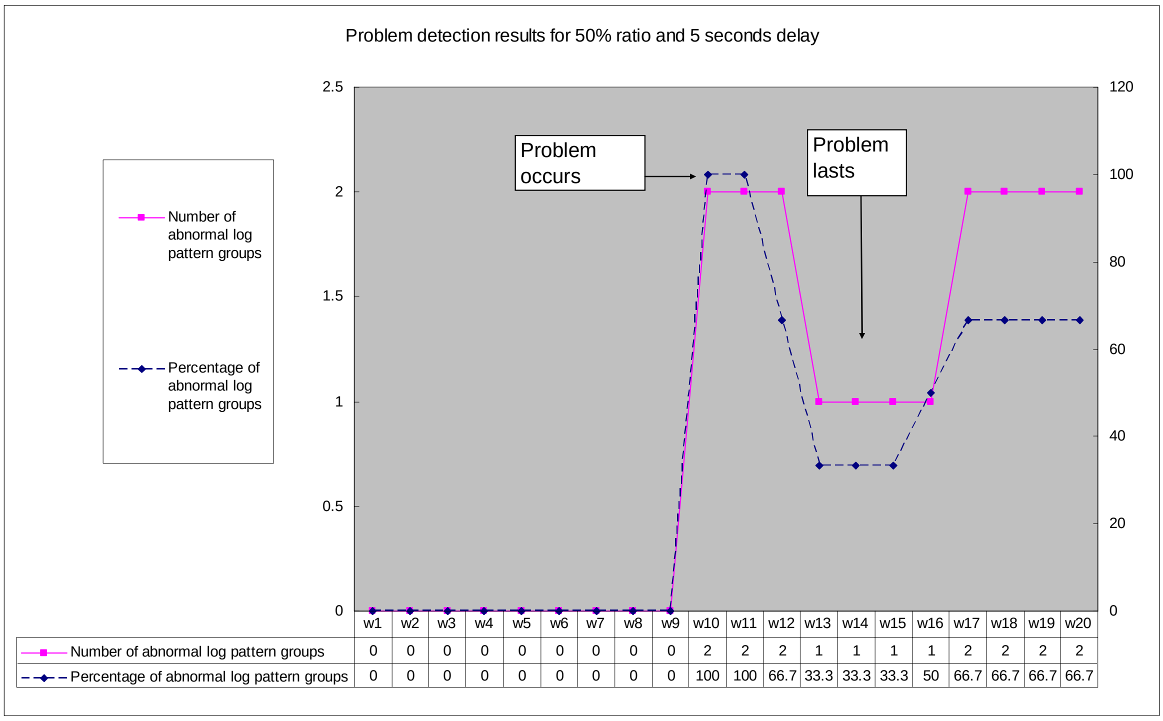


	w 1	w 2	w 3	w 4	w 5	w 6	w 7	w 8	w 9	w 10	w 11	w 12	w 13	w 14	w 15	w 16	w 17	w 18	w 19	w 20
Number of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	2
Number of potential false alarms	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.12: Number of potential false alarms for 25% ratio and 2.5 seconds delay

These potential false alarms are actually caused by a log pattern that indicates the user information can be retrieved from the cache instead of querying the database if the user data has not expired in the memory yet. Since these are not common log patterns that can be observed in sequential N time windows, they are automatically corrected by our sliding window algorithm.

4.6.2 Case 5: 50% Impact Ratio and 5 Seconds Delay



	w1	w2	w3	w4	w5	w6	w7	w8	w9	w10	w11	w12	w13	w14	w15	w16	w17	w18	w19	w20
Number of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	2	2	2	1	1	1	1	2	2	2	2
Percentage of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	100	100	66.7	33.3	33.3	33.3	50	66.7	66.7	66.7	66.7

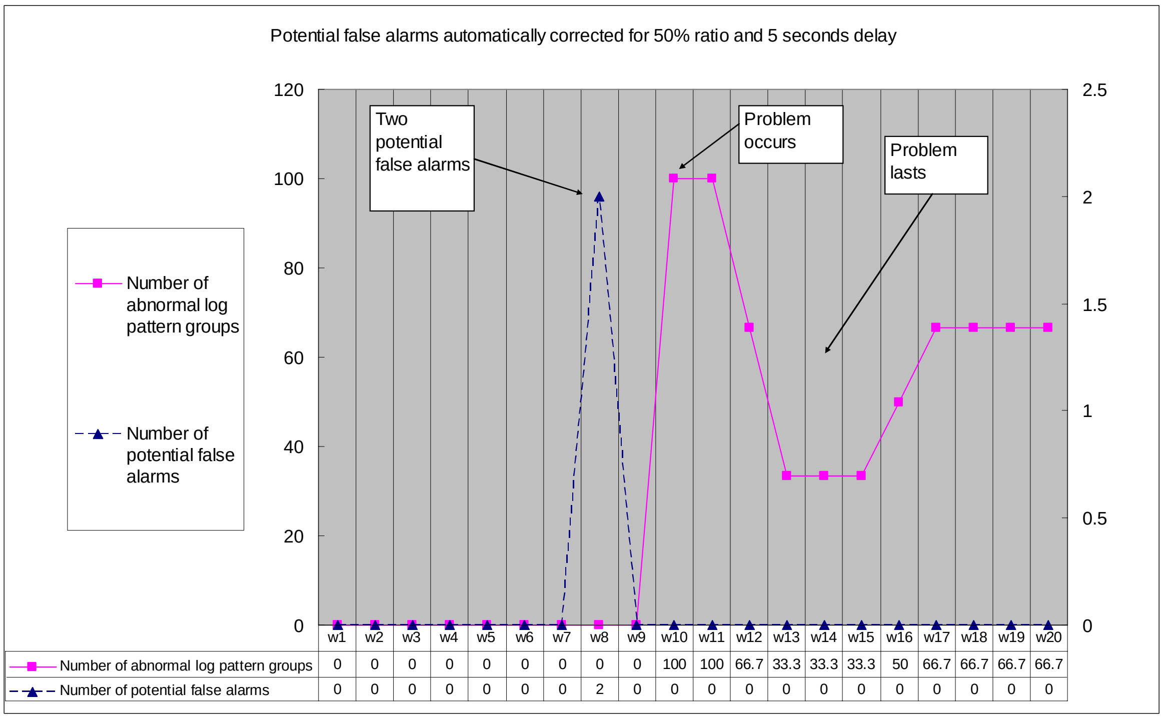
Figure 4.13: Problem detection results for 50% ratio and 5 seconds delay

In this test, the CentralProcess2 connected to the database will regard the injected 5 seconds delay as a symbol that the database is temporarily down and therefore refuse to accept new requests from the GSM Network Simulator. For instance, as shown in Table 4.15, the number of pattern 1 in the MFH group decreases to a very small number starting from the time window between 9:01 and 10:01. On the contrary, the total number of MTH messages remains almost the same, but the distribution of messages in two bins changes significantly after the problem occurs. If we compare the number of pattern 2 in BIN1 and BIN2 before and after the time window 9:01, we can see that the total number of log pattern 2 does not change significantly. However, the distribution of messages in two bins is reversed, which can be also detected by our chi-square goodness of fit test.

Time window	Pattern1_bin1	Pattern1_bin2	Pattern2_bin1	Pattern2_bin2
6:01	102	2	56	4
7:01	96	8	47	13
8:01	91	2	54	6
9:01	**1**	**3**	**15**	**50**
10:01	2	3	33	61

11:01	2	1	31	62
12:01	6	0	31	64
13:01	4	0	30	64

Table 4.15: Messages divided into bins based on processing time



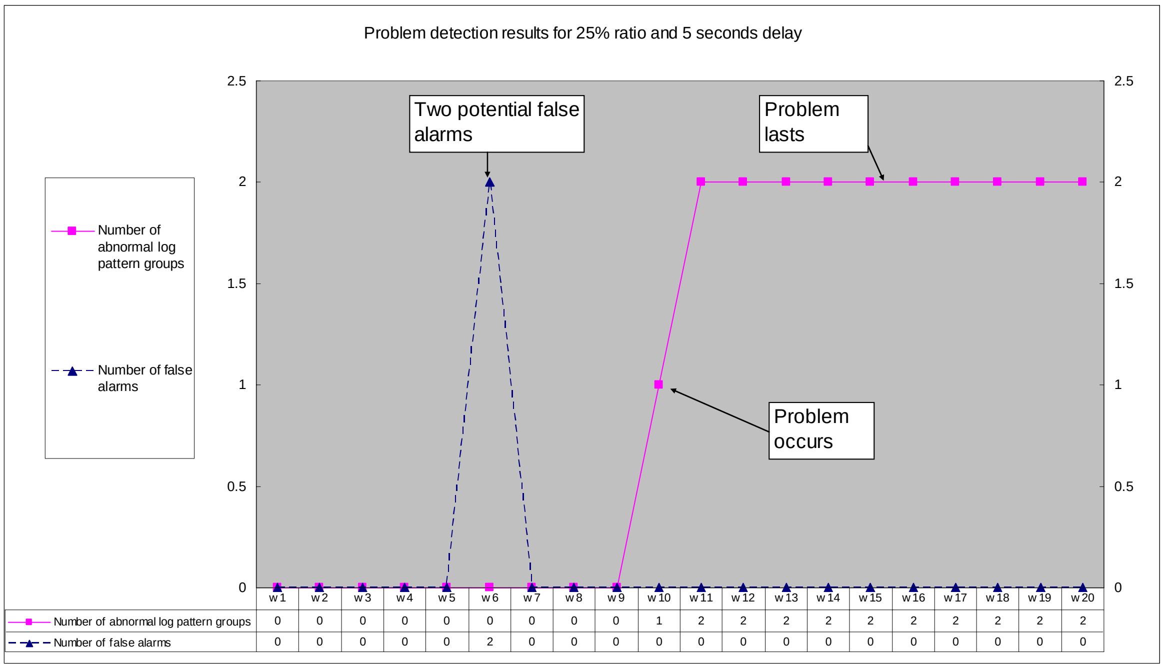
	w1	w2	w3	w4	w5	w6	w7	w8	w9	w10	w11	w12	w13	w14	w15	w16	w17	w18	w19	w20
Number of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	100	100	66.7	33.3	33.3	33.3	50	66.7	66.7	66.7	66.7
Number of potential false alarms	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.14: Potential false alarms corrected by our sliding window approach

Again, two potential false alarms are automatically corrected by our sliding window approach.

4.6.3 Case 6: 25% Impact Ratio and 5 Seconds Delay

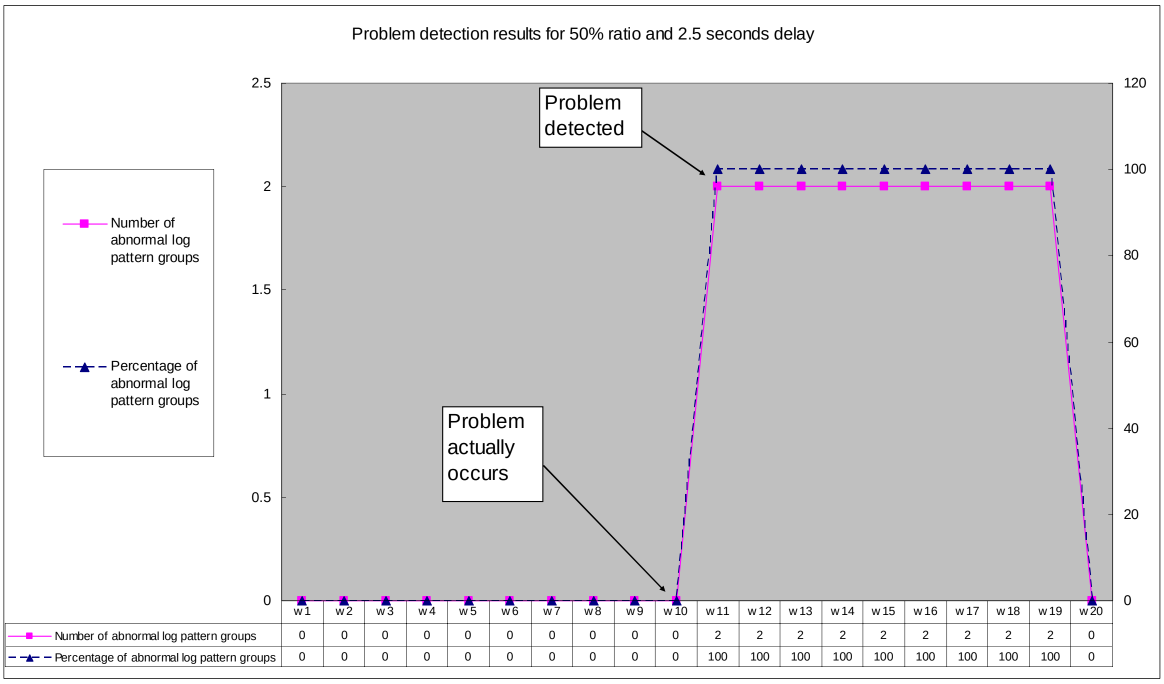
The test results and the root cause of the problems in this log data set are identical to the case 5, so we do not give further analysis here.



	w1	w2	w3	w4	w5	w6	w7	w8	w9	w10	w11	w12	w13	w14	w15	w16	w17	w18	w19	w20
Number of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	1	2	2	2	2	2	2	2	2	2	2
Number of false alarms	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.15: Problem detection results for 25% ratio and 5 seconds delay

4.6.4 Case 7: 50% Impact Ratio and 2.5 Seconds Delay



	w1	w2	w3	w4	w5	w6	w7	w8	w9	w10	w11	w12	w13	w14	w15	w16	w17	w18	w19	w20
Number of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	0
Percentage of abnormal log pattern groups	0	0	0	0	0	0	0	0	0	0	100	100	100	100	100	100	100	100	100	0

Figure 4.16: Problem detection results for 50% ratio and 2.5 seconds delay

The test results and the root cause of the problems in this log data set are almost identical to the case 3. However, after the problem occurs in the 10th time window, our algorithm does not detect the problem immediately. Instead, the problem is detected in the 11th time window. As shown in Table 4.16, there is a change in the number of log patterns in the 10th time window between 9:01 and 10:01. However, such a change is not considered as significant in our chi-square test, so our chi-square accepts that the null hypothesis that the data is normally distributed if compared to the baseline. Since the injected performance impact is set to low level in this test, we think such a delay in the detection result is reasonable. Furthermore, the problem detection result can still help system administrators find the abnormal groups and identify the root cause of the problem.

Time window	Pattern1_bin1	Pattern1_bin2	Pattern4_bin1	Pattern4_bin2
6:01	94	0	25	0
7:01	99	0	21	0
8:01	92	0	28	0
9:01	56	6	11	2
10:01	2	4	1	1
11:01	6	4	0	1
12:01	4	3	2	2
13:01	3	3	2	2

Table 4.16: Messages divided into bins based on processing time

4.6.5 Conclusion

In the last four test cases, we reuse the parameters in the first three tests to perform chi-square goodness of fit over the log data collected from the system where different levels of impact ratio and performance delay are injected. The test results still show an excellent accuracy. Therefore, if our model is properly trained for a particular system, we believe our solution can be used to detect two major types of faults including system errors represented by error event logs and performance issues indicated by abnormal processing time.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we explore a new method of problem determination in message-flow Internet services based on event log analysis. From our work, we can draw the following conclusions.

First, we define the concepts of log events and log patterns in a message-flow Internet service where messages can be correlated by message ID. In such a system, the number of log patterns and the processing time of messages during each time window can be used to describe an accurate statistical model of the system.

Second, we develop a clustering algorithm that combines the “skip-whitespace” algorithm and the SLCT algorithm to identify log events. In addition, we also describe how to identify log patterns by correlating log events with the same message IDs, and develop a grouping algorithm that divides the log patterns into different groups.

Third, we propose a modified version of the chi-square goodness of fit test with a sliding window approach. Our sliding window algorithm can properly determine whether the baseline and the current time window should be shifted or frozen according to the chi-square result. The test results prove that our sliding window approach can significantly reduce the false alarms in our chi-square goodness of fit test. Moreover, we also provide a visualization diagram to indicate the number and the percentage of abnormal log pattern groups. With the help of such system status information, developers and administrators can identify the root causes of problems in a more straightforward way.

Last, we evaluate our solution with seven different types of log data sets collected from the BIS engine, including system errors and application errors. According to the test results, our solution shows promising accuracy.

From the discussion above, we conclude that our solution is a feasible and efficient solution for problem determination in message-flow Internet services.

5.2 Future Work

There are still some potential extensions for our work, which include:

1. Though the test results in our simulation environment show high accuracy, we have not finished the evaluation with the log data sets collected from our internal test environments or even a production environment. The complexity of message-flow systems and the nature of syslog mean that unlimited formats of text descriptions could introduce many potential problems in the log event clustering and log pattern clustering.
2. The parameters used in our algorithm are either hardcoded or determined according to our performance requirement for the test environment. In the future, we want to use machine learning to determine the proper values for these parameters automatically.
3. Some fields in the syslog have not been used in our approach. For instance, the values of the log severity field such as debugging, warning, and critical contain very useful information about the system status. We could add these fields to the input of our chi-square algorithm and give it a higher weight in the test.
4. There is much practical work left for other Internet services, especially the systems without the concept of message ID. For example, we would like to apply our framework to the Apache access logs and see if our solution could still correctly determine problems. A possible change would be that in Apache access logs, a log event can be equal to a log pattern since there is no concept of message ID and thus there would be no correlation by message ID. Our expectation is that our chi-square goodness of fit test with the sliding window approach can still be applied to the statistical data of the Apache event logs.

References

[1] IBM Corporation. LTA (Log and Trace Analyzer) Version 1.0.1, manual. IBM Corporation, 2003.

[2] Rodrigo Fonseca George Porter Randy H. Katz Scott Shenker Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. 1999.

[3] Mike Y. Chen, Emre Kıcıman*, Eugene Fratkin*, Armando Fox*, Eric. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. 2002.

[4] Research In Motion. BIS (Blackberry Internet Service), http://na.blackberry.com/eng/services/internet/. 2009.

[5] Stephen E. Automated System Monitoring and Notification With Swatch. 1993.

[6] Risto V. SEC- A lightweight event correlation tool. 2002.

[7] Jon Stearley. Towards Informatics analysis of syslog. 2004.

[8] Risto Vaarandi. A Data Clustering Algorithm for Mining patterns From Event Logs. 2003.

[9] Andreas Wespi, Marc Dacier and Herve Debar. An Intrusion Detection System based on the Teiresias Pattern Discovery Algorithm. 1999.

[10] Sivan Sabato. Analyzing system Logs: A New View of What Is Important (event ranking). 2007.

[11] Sheng Ma and Joseph L. Hellerstein. Mining Partially Periodic Event Patterns with Unknown Periods. 2001.

[12] Tao Li, Feng Liang, Sheng Ma. An Integrated Framework on Mining Logs Files for Computing System Management. 2005.

[13] Yasuo Musashi, Ryuichi Matsuba. Statistical Analysis in Log Files of SMTP server and DNS. 2003.

[14] J.L.Hellerstein, S.Ma, C.-S.Perng. Discovering Actionable Patterns in Event Data. 2002.

[15] James E. Webviz: A Tool for World Wide Web access log analysis. 1994.

[16] Dan Gunter, Brian Tier. NetLogger: A Toolkit for Distributed System Performance Analysis. 2000.

[17] Wikipedia. Degrees of freedom, http://en.wikipedia.org/wiki/Degrees_of_freedom_(statistics). 2009.

[18] Wikipedia. Pearson’s chi-square test, http://en.wikipedia.org/wiki/Pearson%27s_chi-square_test. 2009

[19] Wikipedia. Goodness of fit, http://en.wikipedia.org/wiki/Goodness_of_fit. 2009.

[20] J. Case. RFC1157: Simple Network Management Protocol (SNMP), http://www.ietf.org/rfc/rfc1157.txt. 1990.

[21] J. L. Hellerstein, S. Ma, and C. Perng. Discovering actionable patterns in event data,. IBM Systems Journal, vol. 41, no. 3, 2002.

[22] IBM. Tivoli Business Systems Manager, 2001, http://www.tivoli.com

[23] Common Base Event , http://www.ibm.com/developerworks/library/specification/ws-cbe/

[24] Ruoming Jin   Breitbart, Y.   Muoh, C.   Data Discretization Unification, 2007

[25] Sotiris Kotsiantis, Dimitris Kanellopoulos. Discretization Techniques: A recent survey. 2006.

[26] C. Lonvick. RFC 3164 - The BSD Syslog Protocol. 2001.