# Failure Categorization for Problem Diagnosis on Exception-Based Software Systems

Shuhai Li
School of Computer Science and Engineering
Beihang University
Beijing, China
e-mail: SebertHai@gmail.com

***Abstract*—Traditionally, distributed system software developers print log messages when creating a program to track the runtime status of a system to help identify where problems may have occurred while the program is running. People often use system logs produced by distributed systems for troubleshooting and problem diagnosis. However, there may be thousands of failed jobs occurring within a short time. Manually inspecting these jobs one by one to detect anomalies is unfeasible due to the increasing scale and complexity of distributed systems. Since many failed jobs may have the same cause, there is a great demand for automatic job categorization techniques based on log analysis to help developers prioritize job investigation. Described herein is an unstructured log analysis technique for job categorization. In the technique, we propose a novel algorithm to categorize log messages into different categories without heavily relying on application specific knowledge, based on which jobs can be categorized.**

***Keywords-log analysis; job clustering; message categorization; problem diagnosis***

## I. INTRODUCTION

Large scale distributed systems are becoming key engines of IT industry. For a large commercial system, execution anomalies, including erroneous behavior or unexpected long response times, often result in user dissatisfaction and loss of revenue. These anomalies may be caused by hardware problems, network communication congestion or software bugs in distributed system components. Most systems generate and collect logs for troubleshooting, and developers and administrators often detect anomalies by manually checking system printed logs. However, as many large scale and complex applications are deployed and many failed jobs may occur within a short time, manually detecting anomalies becomes very difficult and inefficient. At first, it is very time consuming to diagnose through manually examining a great amount of log messages produced by a large scale distributed system. Secondly, a single developer or system administrator may not have enough knowledge of the whole system, because many large enterprise systems often make use of Commercial-Off-the-Shelf components (e.g. third party components). In addition, the increasing complexity of distributed systems also lowers the efficiency of manual problem diagnosis further. Therefore, developing automatic job categorization tools becomes an essential requirement of many distributed systems to help developers prioritize job investigation and ensure the Quality of Service.

Assumptions: When a failed job occurs, its corresponding log file always contains some exception messages which may partly explain the causes of the anomaly. We name these exception messages as TRAP messages. In the technique, job categorization is based on the TRAP messages in the corresponding log files. We assume that each log message has a corresponding time stamp that indicates its generation time. We further assume that the logs are recoded using thread IDs or request IDs to distinguish logs of different threads or work flows. Most modern operating systems (such as Windows and Linux) and platforms (such as Java and .NET) provide thread IDs. We can therefore work with sequential logs only. In addition, we assume that each log entry contains a job ID property, which records which job the log entry belongs to.

This technique contains the following steps: exception message preprocessing, exception message categorization by log key, and job categorization by job signature. Besides, during exception message categorization, we can do exception message category significance evaluation and key object analysis.

## II. EXCEPTION MESSAGE PREPROCESSING

As in a distributed system, many workflows may run simultaneously for a job, the log file generated by the corresponding job probably contains interleaving log messages of different work flows. What's more, the log file of a job may contain some log messages that do not actually written by this job. Then these log messages are noisy messages of the log file. In addition, an exception of a job often triggers other exceptions, which will also be written in the corresponding log file. Then these exception messages are actually redundant messages of its log file. Therefore, it is necessary to do log messages preprocessing of original log files. We take PowerShell logs for example.

### *A. Remove noisy messages*

A log file of a job may contain some noisy messages that actually should not belong to this job. This is because in a distributed system, many jobs run simultaneously and if any log entry does not record the correct job ID, it will probably be written into a log file which is actually another job. An example is as "TABLE I".

Published by Atlantis Press, Paris, France.

TABLE I. An example of a log file

| **Process Id** | **Thread Id** | Job Id | Message | **Machine Name** |
|---|---|---|---|---|
| *6496* | *25* | *10551060* | *Execute job for execution.* | *Machine Name1* |
| *7860* | *4* | *-1* | *The job has been ready.* | *Machine Name2* |
| *6496* | *25* | *10551060* | *Updating build version.* | *Machine Name1* |
| *6496* | *25* | *-1* | *TRAP: … …* | *Machine Name1* |
| *7860* | *4* | *-1* | *TRAP: … …* | *Machine Name2* |

"TABLE I" is part of log entries of a log file "*10551060.log*". Of the properties in the log, "*ProcessId*", "*ThreadId*" and "*MachineName*" together distinguish different workflows. In the example above, there are two different threads: "*6495, 25, MachineName1*" and "*7860, 4, MachineName2*".

The noisy message removing rule is that if no log entry of a certain work flow contains the correct job ID that is the same with the job name without extension, then we consider the messages of this thread as noisy messages. In the example above, all *JobId*s of "*7860, 4, MachineName2*" workflow are "*-1*", then we consider the messages of "*7860, 4, MachineName2*" work flow as noisy messages and they should be removed.

After the processing, the remaining log entries are as "TABLE II".

TABLE II. Log entries after removing noisy messages

| **Process Id** | **Thread Id** | Job Id | Message | **Machine Name** |
|---|---|---|---|---|
| *6496* | *25* | *10551060* | *Execute job for execution.* | *Machine Name1* |
| *6496* | *25* | *10551060* | *Updating build version.* | *Machine Name1* |
| *6496* | *25* | *-1* | *TRAP: ……* | *Machine Name1* |

*B. Remove redundant TRAP messages*

After removing noisy messages, we need to remove redundant TRAP messages of the log file. For example, a log file has the following TRAP messages after removing noisy messages:

a. *TRAP: Retry exception: [MACHINE: MachineName1:* ***TRAP: Exception [Exception:Failed to adjust User Rights permissions. TRAP_DETAILS: ErrorFile [C:\ Common\Collection.ps1], ErrorLine [1495: throw $errorRecord]. TRAP_ACTION: Rethrow.****]. TRAP_ACTION: Job 1024232 will retry.*

b. ***TRAP: Exception [Exception:Failed to adjust User Rights permissions. TRAP_DETAILS: ErrorFile [C:\ Common\Collection.ps1], ErrorLine [1495: throw $errorRecord]. TRAP_ACTION: Rethrow.***

We can find that the longer TRAP message contains exactly the shorter TRAP message. We consider the shorter TRAP as the root exception of the job and consider the longer TRAP message as redundant TRAP message.

The redundant TRAP message removing rule is that if a certain TRAP message, recorded as "Trap1", contains any one TRAP message of this log file whose length is shorter than the length of "Trap1", then we consider "Trap1" as a redundant TRAP message and "Trap1" needs to be removed from the log file.

After removing redundant TRAP messages, the example above has the remaining TRAP:

*TRAP: Exception [Exception:Failed to adjust User Rights permissions. TRAP_DETAILS: ErrorFile [C:\Common\Collection.ps1], ErrorLine [1495: throw $errorRecord]. TRAP_ACTION: Rethrow.*

*C. Replace nested TRAP with its inner layer TRAP*

After removing redundant TRAP messages of a log file, if there are still nested TRAP messages remained, we need to replace them with their inner layer TRAP messages. For example, a log file has the following TRAP message after removing redundant messages:

*TRAP: Retry exception: [MACHINE: MachineName1:* ***TRAP: Exception [Exception:Failed to adjust User Rights permissions. TRAP_DETAILS: ErrorFile [C:\Common\Collection.ps1], ErrorLine [1495: throw $errorRecord]. TRAP_ACTION: Rethrow.****]. TRAP_ACTION: Job 1024232 will retry.*

We can find that the TRAP message is a nested one. We consider the inner layer TRAP as the root cause of the TRAP and replace the TRAP with its inner layer TRAP. Accordingly, the rule of this process is that if a TRAP message is a nested one, we replace the message with its inner layer TRAP.

After replacing nested TRAP with its inner layer TRAP, the example above has the remaining TRAP:

*TRAP: Exception [Exception:Failed to adjust User Rights permissions. TRAP_DETAILS: ErrorFile [C:\Grid\Common\Collection.ps1], ErrorLine [1495: throw $errorRecord]. TRAP_ACTION: Rethrow.*

## III. EXCEPTION MESSAGE CATEGORIZATION

Although there may be a considerable amount of exception messages in failed job logs, probably many of these exception messages are printed out by the same source code statement. We consider the messages that are printed out by the same source code statement as the same category.

If we categorize the exception messages into different categories, we can further extract parameters of each exception messages. These parameters may point out the object that goes wrong, which may be a website, a server ID, etc. We also take PowerShell logs for example to explain how to do exception message categorization.

### *A. TRAP message categorization by edit distance*

a. Calculate edit distances of each two TRAP messages

Given two character strings and, the edit distance between them is the minimum number of edit operations required to transform into. Most commonly, the edit operations allowed for this purpose are: (i) insert a character into a string; (ii) delete a character from a string and (iii) replace a character of a string by another character. Herein we take a word of an exception message as a character to calculate the edit distance. For example, the edit distance of the string "I am David" and "I am John" is 1, not 5. We take a word as an edit distance unit because the edit distance calculated in this way can better reflect the degree of similarity of two exception messages. After this step, we can get the edit distances of each two TRAP messages.

b. Analyze these edit distances and set a threshold value

We need to set a threshold value of edit distances to judge whether any two TRAP messages are of the same category. That is, the edit distance between each pair of exception messages in the same group should be smaller than a distance threshold. The idea of automatically determining the threshold is that Intra-class distances and Inter-class distances should be quite different. Based on such idea, we use Gaussian mixture model to obtain the threshold.

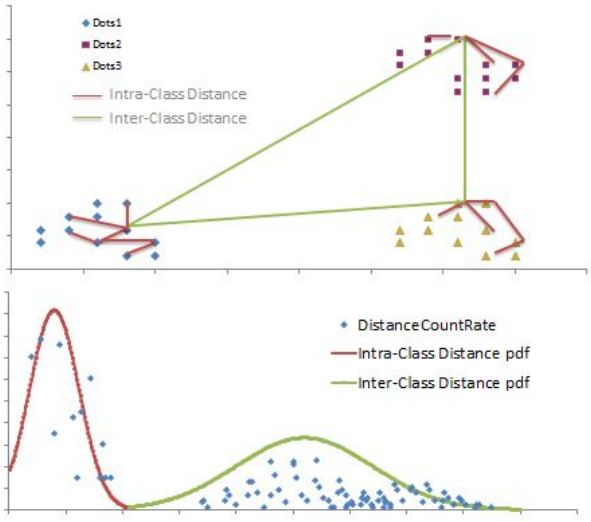


Figure 1. Gaussian mixture model

As "Fig. 1", there are three groups of exception messages. However, we do not know which ones belong to the same group in advance. Next, we compute distance of each pair of messages, and obtain a set of distance. We use the Gaussian mixture model to fit the distance distribution. The left and right Gaussian components correspond to intra-class distance and inter-class distance. Therefore, we can automatically determine the threshold. (Refer to "(1)")

$$S_i \in G; if\ \forall S_j \in G, D(S_i, S_j) \leq Th \tag{1}$$

*(Th is the threshold of distance)*

It should be pointed out that different kinds of exceptions usually have very different messages. They can usually be discriminated with high accuracy.

c. Categorize TRAP messages depending on the threshold value

If the edit distance of two TRAP messages is lower than the threshold value set in the last step, then we consider these two TRAP messages are of the same category, through which we can categorize TRAP messages into different categories.

Herein we give a real example to show the process of TRAP message categorization by edit distance.

The edit distances of each two TRAP messages are as follows.

| | Trap1 | Trap2 | Trap3 | Trap4 |
|---|---|---|---|---|
| Trap1 | 0 | | | |
| Trap2 | 1 | 0 | | |
| Trap3 | 15 | 16 | 0 | |
| Trap4 | 13 | 15 | 2 | 0 |

⇓

We sort the edit distances above: 1, 2, 13, 15, 15, 16. We can find that there is a great gap between the higher values and the lower ones. We use the Gaussian mixture model to set the threshold value as 5.

⇓

If the edit distance of two TRAP messages is smaller than the threshold value, then they are of the same category.
So the result of categorization of the example above is:
1st category: Trap1, Trap2;
2nd category: Trap3, Trap4.

### *B. Common sub-string extraction*

After the automatic TRAP message categorization, we need to extract the common sub-strings of each category before parameter extraction.

a. Extract the preparatory longest common sub-string of each category

This is implemented by LCS standard algorithm. We also take a word of a string as the unit for the LCS algorithm. For example, the longest common sub-string of "Failed to connect database server SQL020" and "Failed to connect database server SQL023" is "Failed to connect database server".

b. Extract the accurate longest common sub-string of each category

After extracting the preparatory LCS by the LCS algorithm, there may also be potential parameters in the preparatory LCS. We need to remove these parameters in

LCS and get the accurate longest common sub-string of each TRAP message category. It is necessary to write a configuration file that stores the possible parameter patterns. And we compare the preparatory LCS with the parameter patterns in the configuration file. We remove the sub-strings that can match any of the parameter patterns and get the accurate LCS.

c. Extract the common sub-strings of each category by LCS

After we get the accurate LCS, we compare the LCS with each TRAP message of the category. Then we can extract the common sub-strings separated by the possible parameters and the different parts of TRAP message of this category. The common sub-strings can be used to extract parameter of each TRAP message.

### C. *Parameter extraction*

After we get the common sub-strings of each category, we can extract the parameters of each trap message.

a. Extract the possible parameters by comparing each trap message with common sub-strings

We consider the string between each two neighbor common sub-strings as a possible parameter.

b. Compare each possible parameter with parameter patterns in configuration file

If the possible parameter matches a parameter pattern, then the substring that exactly matches the pattern is printed out. Otherwise, we take the whole parameter string as a parameter.

## IV. JOB CATEGORIZATION

Each failed job log file probably contains some exception messages. All exception messages of all failed job log files are categorized after exception message categorization. If a job log file contains the same set of exception message category with another one, then the two job log files are of the same job category. We call the set of exception message category of each job as "job signature". We consider the jobs whose job signatures are the same as the same job category.

For example, if we have got the job signature of each job:

- Job1:
  {TRAP category1, TRAP category2, TRAP category3}
- Job2:
  {TRAP category4, TRAP category3, TRAP category1}
- Job3:
  {TRAP category3, TRAP category4, TRAP category1}
- Job4:
  {TRAP category2, TRAP category1, TRAP category3}

We can see that Job1 and Job4 have the same job signature. In addition, Job2 and Job3 have the same job signature. So finally we can categorize these job logs into several job categories:

- Job Category1: {Job1, Job4}
- Job Category2: {Job2, Job3}

## V. EXCEPTION MESSAGE CATEGORY SIGNIFICANCE EVALUATION

Exception message category significance can be evaluated by the following standards.

- Job occurrence number
  - The total number of jobs where a message of the exception message category occurs.
- Job category occurrence number
  - The total number of job categories where a message of the exception message category occurs.
- Message occurrence number
  - The total number of message occurrence that belongs to the exception message category.

## VI. KEY OBJECT ANALYSIS

Key object parameter patterns are predefined in a configuration file, such as database server "*SQL024*", application server "*App007*". These parameters may occur in different job log files. By doing key object statistics, we can get job distribution situation on corresponding parameters.

We get the parameter statistics from the step of parameter extraction.

The parameter statistics results of some actual provision logs are as "TABLE III".

TABLE III. The top 3 suspicious "database server"

| **Database server name** | **Number of suspended jobs** | **Percentage in all jobs** |
|---|---|---|
| *SQL027* | *218* | *24%* |
| *SQL021* | *33* | *4%* |
| *SQL023* | *32* | *4%* |

## REFERENCES


[1] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In POPL'08.

[2] NetApp. Proactive health management with auto-support. NetApp White Paper, 2007.

[3] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In OSDI'04.

[4] Dell. Streamlined Troubleshooting with the Dell system E-Support tool. Dell Power Solutions, 2008.

[5] Hadoop. http://hadoop.apache.org/core.

[6] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "SALSA: Analyzing Logs as State Machines", In the proceeding of 1st USENIX Workshop on the Analysis of System Logs, Dec. 2008.

[7] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosingproduction run failures at the user's site. In SOSP'07.

[8] S. Ghemawat and S. Leung, "The Google File System", In the proceeding of ACM Symposium on Operating Systems Principles (SOSP), Oct. 2003.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In PLDI '88.

[10] W. Jiang. Understanding storage system problems and diagnosing them through log analysis. Ph.D. Dissertation.