THE UNIVERSITY OF CHICAGO

CLUSTERING AND RECOGNIZING JOB FAILURES IN LARGE-SCALE DISTRIBUTED SYSTEMS

A DISSERTATION SUBMITTED TO THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES IN CANDIDACY FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY JING TIE

CHICAGO, ILLINOIS AUGUST 2009
Abstract

Large-scale distributed systems are widespread, supporting millions of concurrent users from various application areas. This type of infrastructure provides common, friendly and flexible interfaces, but also is subject to many dynamic failures. The large size and complexity of these systems makes diagnosing failures a challenging problem. This paper focuses on learning and recognizing job failures automatically.

First, we study the characteristics of job failures in a large-scale distributed system PanDA (distributed production and distributed analysis system for ATLAS experiments at LHC). Based on two-year records, we have some interesting findings. One is that job failures are both temporally and spatially correlated, and strongly temporal correlated failures are likely to have the same root cause. Bag-of-Tasks applications and their jobs arrival pattern - similar jobs arrive in groups - are big contributors to this phenomena.

Based on the findings above, we develop a mechanism for discovering the patterns of different failures, and recognizing similar ones in the future. Our mechanism learns different correlation models between job metrics and job statuses using Bayesian network classifiers, and then extracts a signature for each job based on the best model. It also clusters the similar signatures together, which helps labeling the signatures with different root causes. Then the next time a failure arrives, it can be mapped to the most likely root cause by retrieving from all the labeled signatures. The mechanism continues learning new models using a sliding window, and includes good models to update the ensemble of models.

Our experimental results show that the models learned have very high accuracy, high hit rate, and low false alarm rate in predicting job statuses in the near future. By analyzing the centroid signatures in the generated clusters of failures, we illustrate that those clusters can indeed differentiate or indicate meaningful root causes of failures.
Acknowledgement

First of all, I would like to gratefully and sincerely thank my advisor, Professor Ian Foster, for his guidance, support, understanding, and patience. It is really a great fortune for me to work with him. His insightful opinions and nimble thoughts have always been grappling the crucial points and motivating me during my research. He also gives me the freedom to explore my own interest, and at the same time, offers invaluable advices, helps and opportunities.

I am also deeply grateful to Professor Michael O’Donnell and Professor Svetlozar Nestorov, for serving on my thesis committee, and offering me constructive comments on this work. Thank Prof. Anne Rogers and Prof. Matei Ripeanu for many great advices. I believe that I will learn a lot from them and what I learn will benefit all my life.

I would also like to give my thanks to Marco Mambelli. As a troubleshooter on PanDA, he has been helping me with all kinds of problems that I met during the experiments, including providing the dataset, explaining execution mechanisms of PanDA, and analyzing the learning results. And I appreciate all the help and suggestions from the local OSG group members, Alina Bejan, Rob Gardner, Suchandra Thapa and Michael Wilde. I would also like to thank Paul Nilsson, Torre Wenaus, Aaron Thor, and Yuri Smirnov from PanDA team; they offered me PanDA job archive, and answered questions about the dataset. This work could not have been completed without their help.

I would like to thank all people in Computer Science Department, Computational Institute at University of Chicago. In particular, I would like to thank my colleagues Xuanyuan Zhou, Quan Tran Pham, Ioan Raicu, Zhao Zhang, Lingyun Yang, Gabri Turcu, Xinhua Shi, Allan Espinosa, Yi Zhu, Soner Balkir, Yong Zhao, and many others whose names not listed here. I cherish all the discussions and talks with you.

Many thanks also go to my friends at the University of Chicago, and in China. Your support means a lot to me!

Last but not least, I would like to thank my parents, who love me with all their hearts, and sacrificed so much to provide me with a lot of opportunities in my education and life. They are also my best friends, who always standing by my side and encouraging me to pursue my dreams.
Abstract.....................................................................................................................................ii
Acknowledgement ............................................................................................................... iii

1 Introduction ................................................................................................................................1
  1.1 Problem definition...............................................................................................................1
  1.2 Failures in large-scale distributed systems .................................................................2
  1.3 Current diagnostic techniques .....................................................................................3
  1.4 Challenges of failure diagnosis ...................................................................................4
  1.5 Background on classification and clustering ............................................................5
  1.6 Learning and recognizing job failures automatically .................................................6

2 Job Failure Characteristics in PanDA...................................................................................7
  2.1 PanDA Overview............................................................................................................7
  2.2 Job Failures Distribution ..........................................................................................9
  2.3 Similar jobs arrive in groups ....................................................................................12
  2.4 Temporal Correlation of failures ...............................................................................13
  2.5 Spatial Correlation ....................................................................................................15

3 Clustering and Recognizing Job Failures ...........................................................................17
  3.1 Overview of the mechanism .......................................................................................17
  3.2 Assumption of building classifier ..............................................................................18
  3.3 Building correlations between job metrics and job statuses ....................................19
  3.4 Updating ensemble of models ...................................................................................20
  3.5 Extracting signatures ................................................................................................22
  3.6 Clustering signatures ................................................................................................23
  3.7 Retrieving Signatures ...............................................................................................23
Chapter 1

Introduction

The trend of collaborative computing among distributed computers is growing rapidly in recent years, not only in size and variety of applications, but also in scale and complexity of working environment. Open Science Grid (OSG) [27], a national distributed computing grid for data intensive research, has been supporting research in physics, astrophysics, and bioinformatics and so on for years. Commercial tools, such as Amazon Elastic Compute Cloud (Amazon EC2) [28], allow developers to execute jobs across hundreds of machines easily at a low price. All these systems shield low-level complexity from users, and provide common, friendly infrastructures. However, as a result of different user requirements, various tools and components have and will be deployed on those systems, and these components communicate with each other in many different ways. It is clear that job failures are inevitable, since any problem on any component might lead to the final failure. And they are also not easy to diagnose due to the system size and complexity. Thus troubleshooting these problems in an efficient way is becoming a big challenge in large-scale distributed systems.

1.1 Problem definition

First of all, let us introduce some concepts in failure diagnosis [13]:

**Event** is defined as an exceptional condition occurring in the operation of a system.

**Faults** (also called as root causes or problems) are a class of events that can cause other events but not themselves caused by other events.

**Error** is a discrepancy between a computed, observed, or measured value or condition and a true, specified, or theoretically correct value or condition. Error is a consequence of a fault.

**Failure** is an error that is visible externally, occurring when a delivered service deviates from the specified service. In particular, job failure occurs when a job does not generate the correct result. Errors invisible to the outside world are not easy to observe, but may propagate within the system and cause failures. Fault is a root cause of failure.

**Symptoms** are external manifestations of failures. They are metrics observed from monitoring
and logging systems.

*Failures Diagnosis* (also called *fault localization*) is a process of deducting the fault of a failure from the symptoms.

### 1.2 Failures in large-scale distributed systems

Failures in distributed systems are caused by four categories of faults or their combinations:

- **User mistake (and application bug):** inappropriate user operations, e.g. using expired proxy, job description mistakes, or bugs in application code;
- **Middleware fault:** software bugs inside or between different components, e.g. mismatched or missing interfaces, job manager not responding;
- **Configuration fault:** system initializing/maintenance problems, e.g. incorrect authorization;
- **Hardware fault (and network fault):** e.g. machine down, power outage, connection closed, packet delay or missing.

![Figure 1.1: Percentages of positive responses of the most frequent failures on grids (left) and the greatest problems for recovering from a failure (right) in 2003 survey [17]](image)

Figure 1.1: Percentages of positive responses of the most frequent failures on grids (left) and the greatest problems for recovering from a failure (right) in 2003 survey [17]

Grids are widespread large-scale distributed systems [22], thus we choose them as our major objects in the paper. The left graph in Figure 1.1 shows results from a 2003 survey on grid users: configuration and middleware faults are the top two most frequent problems. One explanation for this phenomenon is that in order to support different user requirements on highly dynamic environments, the systems have to update software, configurations and user accounts frequently; and it takes time to coordinate them. For example, for one of the most frequent failures - user identification problems, OSG must ensure all the following components work appropriately for
the user: (1) user proxy is valid; (2) the work site is running; (3) the site recognizes the CA (certificate authority) that granted the user’s certificate; (4) the CA is not expired on the site; (5) user’s clock is synchronized with the site's; (6) the site can authorize the user [25]. And usually, all these components have different configurations, and sometimes even different mechanisms for different sites and users.

When users encounter failures on grids, the greatest problem is diagnosing the failures; 71% of the user responses in the survey report the problem as shown in the right graph of Figure 1.1. Failures caused by different faults behave differently, but sometimes they appear very similar in the final results. This situation makes troubleshooting extremely difficult. Take “staging input files error” for instance, it is unclear whether it is caused by a data transfer problem (middleware), or user work directory limited right (configuration), or network disconnection (network) or writing to disk failure (hardware), without examining the logs from all these components.

As we can see, job failures in large-scale distributed systems are difficult to diagnose. Then a lot of logging and monitoring services are set up, to capture the abnormalities from the job traces.

1.3 Current diagnostic techniques

Recently, there is rising research on finding characteristics of failures using data mining and machine learning algorithms [3, 4, 7-10, 12, 15, 18, 19], but for most practical systems, analyzing logs and probing tests by troubleshooters are still the major or even the only techniques available. Experienced troubleshooters can figure out the root causes of some failures, but these techniques have obvious limitations:

- Duplicate work: when similar problems happen again on different sites, for different jobs and users, troubleshooters have to examine everything almost from scratch to identify the root causes, even though all the information needed might have already been collected;
- Little experience on new or hidden problems: if troubleshooters have not seen the problems before, they may not know where to start, since the final return information is too coarse to reduce the large number of the possible candidates;
- Not efficient: usually diagnosing failures is group work. One troubleshooter has to wait for others’ responses because of limited rights or experience;
Having experienced troubleshooters takes a lot of time and money.

These issues are mostly due to human labor limitations. They can be improved if we could use machines to extract the features of different failures from the logs, remember these features, and recognize the root cause by retrieving from the existing features.

1.4 Challenges of failure diagnosis

There are countless possible faults existing in the systems, which are the products of the natures of large-scale distributed systems: diversity and dynamic development. It is known that diagnosing one workstation has already been nontrivial, so we can imagine the difficulty for a dynamically changing system across hundreds of organizations.

First, let us look at the four layers of metrics [24]. These metrics could be regard as symptoms of the failures. Take job execution for example:

- **Application-level metric:** e.g. job description (id, name, and code), user id, input and output files (name, size, location, format, etc), job status, and job start/end time.
- **Middleware-level metric:** the metrics of all the components that supports job executing. E.g. security module (authentication, authorization), data transfer service (server status, version, etc), scheduler (status, queues, version, etc), computing element (gatekeeper status, worker nodes statuses, configuration, queues, etc), storage management (server status, replication policy, logical/physical name mapping, etc).
- **Network-level metric:** e.g. network connection status, data transfer rate between all different components.
- **Resource-level metric:** e.g. CPU utilization, memory/storage usage, packet I/O rates of all the machines in all the related components.

If we consider the metrics of all the layers and components together, it will be a huge amount of data, too difficult and time-consuming to work on for human beings. However, most of the time, a fault is only related to a small part of the system. So how to reduce the set of relevant metrics to a humanly manageable one without losing much information is the biggest challenge.
One method is to find the related metrics using data mining and machine learning algorithms. Our work focuses on this method.

Another way is analyzing the metrics vertically: first reduce the metrics to the most relevant components, or network connections between components, and further reduce to the most relevant machines, or configurations, or software. Thus giving strict definitions to the metrics of different layers and finding metric correlations between layers become very challenging.

Combining the above two methods is also a promising method.

1.5 Background on classification and clustering

Large volumes of data sets are emerging as important resources in a wide range of areas, such as high energy physics, bioinformatics, and stock market analysis. Interesting information contained in the data can help leading to new discoveries or making intelligent decisions. But finding out the useful information from the large datasets is very difficult, so people use data mining, machine learning and statistical methods to learn the possible patterns and correlations. In this section, we will introduce the two learning categories: classification and clustering. This work applies both to learn and recognize job failures.

The basic idea of classification is building classifiers on datasets. A dataset contains many instances, and each instance consists of multiple attributes and a label (also called class). A classifier tries to learn a function that maps the selected attributes of each instance to a class as correctly as possible. Once a classifier is built, it can be used to predict new instances. Different algorithms have been developed for learning classifiers in different applications, such as decision trees, Bayesian networks, and Natural networks.

We choose Bayesian network classifiers in this work, which build a graphical model (Directed Acyclic Graph) to represent a joint probability distribution among a group of variables. The classifying process learns a network structure, and then a conditional probability table [16]. When a new instance arrives, the classifier computes the conditional probabilities of the instance belonging to different classes. Naïve Bayesian networks are the simplest and the most common Bayesian networks. They assume that except for the state class, all the other variables are independent of each other and the state class is their only parent. Although this assumption is not
accurate in practice, this type of network has proven to be effective in many situations. Another well-known network is Tree Augmented Naïve Bayesian network (TAN). Each variable in the network, except for the state class, can have at most two parents and one of the parents must be the state class [7].

The other learning category is clustering. It clusters unlabeled instances into groups, while each group contains similar instances. Generally, the algorithm builds a feature vector for each instance, and defines a distance function between two vectors. The grouping process minimizes the average distance from all the instances in a cluster to its centroid. We use K-Medoids, an expanded algorithm of K-Means, as our clustering mechanism [8]. The difference between the two is that the centroid in K-Medoids is an actual instance, while the centroid in K-Means is the weighted average of all the instances in the cluster.

1.6 Learning and recognizing job failures automatically

From previous sections, we know that jobs fail because of countless possible reasons, and diagnosing these failures from so much information is very difficult by current techniques. Thus some research uses data mining and machine learning methods to help determining root causes, and shows promising results. In this paper, we develop the mechanisms previously proposed in [7, 19, 8], and our extensions. It builds classifiers to learn the correlations between job metrics and job statuses on the fly, extracts signatures and clusters them. Then when a new failure occurs, our mechanism points out the similar past failures happened before.

Our work uses the job records collected in PanDA. So the next chapter will introduce job failure characteristics in PanDA. Chapter 3 describes the learning methodology, and then Chapter 4 shows our experimental results. Related work is listed in Chapter 5. Chapter 6 and 7 discuss the future work and conclusions.
Chapter 2

Job Failure Characteristics in PanDA

ATLAS is one of four main particle physics experiments at the Large Hadron Collider (LHC) at CERN (European Organization for Nuclear Research) [31]. PanDA is the distributed production and distributed analysis system for running ATLAS experiments.

PanDA has been running for over three years. Every day, it processes about 50,000 production jobs, and 3,000-5,000 analysis jobs at 100 sites (clusters or High Performance Computer) around the world. Up to January 2009, more than 25 million jobs were executed in the system [26]. The analysis user community counts over 800 scientists and about 300 are heavy/frequent users at any given time. Also, most of the PanDA jobs have been running on OSG. Therefore, PanDA is a good representative of large-scale distributed systems, and the work in this paper is built on its dataset. For this chapter, we first take a look at the whole system and its failure distribution, and then some characteristics of these job failures.

2.1 PanDA Overview

First, let us introduce some fundamental definitions in PanDA [20]:

**Data Block**, an immutable dataset, “is a set of data produced (taken) under the same logical conditions. It is expected to consist of uniform files suitable for further processing with the same application in the transformation chain.” [30] In other words, data files in the same data block are produced by the same executable, and then again used as the input files of another executable.

**Distributed Data Management (DDM)** is the data module for managing input/output data, including transferring and making replicas. The data handling unit in DDM is a data block.

**Computing Element (CE, also called Site)** contains two logical parts: gatekeeper (also called job-manager) and worker nodes. Jobs are distributed to the worker nodes by means of a batch system. Technically, the gatekeeper and the batch system server run on one machine, which is called the CE node, to which a number of separate worker nodes is connected, preferably in a private subnet [32]. Usually CE is a cluster or a high performance computer.
Storage Element (SE) manages the access to the data files. It is usually located close to a CE [32].

Task contains three parts: execution code, input files and output files. Each task has a task id. There are two kinds of tasks: Production task and Analysis task. As shown in the names, production tasks execute different conventional transformations on large quantities of data files, and analysis tasks typically run user-defined testing algorithms [33].

Production tasks are Bag-of-Tasks applications (BOT), parallel applications whose tasks are independent and similar to each other, maybe just using different input files [21]; and one task id represents one BOT application. That is to say, the production tasks with the same task id have the same execution code and different I/O files, but their input (output) files are often on the same SE in PanDA. Analysis tasks, on the other hand, have the same task id - zero, but very different codes and data files.

Job an instance of a task executed at a particular time.

Pilot is a light-weight execution environment implemented in Python which is used to prepare the computing element [23]. After the CE is prepared, the pilot asks for jobs from the server, and executes the jobs on the worker nodes.

Server receives jobs from users through a common interface, assigns them to the CEs, sets up I/O datasets, and dispatch jobs. The server manages all the jobs in PanDA.

Scheduler is the program sending pilots to all the available computing elements.

Figure 2.1: Schematic view of PanDA System
Figure 2.1 illustrates the schematic view of PanDA. Server receives the jobs into a task queue, and assigns them to distributed pilots based on job type, priority, input data and data locality, and the available CPU resources. Once a job is assigned, its data block of the input data are pre-staged to the nearby storage element by DDM; when the data transfer is done, the job in the server queue is ready to be dispatched. Pilots, who are pre-scheduled by scheduler, retrieve the ready jobs as soon as their CPUs become available, and then execute on the worker nodes [20]. Our dataset are the job records from the server.

### 2.2 Job Failures Distribution

![Graph showing job failures and completions from 2006-10 to 2008-08](image)

**Figure 2.2:** Number of Jobs failed and finished from 2006-10 to 2008-08

![Graph showing monthly job failure rates from 2006-10 to 2008-08](image)

**Figure 2.3:** Failure rates per month from 2006-10 to 2008-08
In the job records from October 2006 to August 2008, there are more than 16 million jobs, 4.6 million of which are failures. Figure 2.2 displays the total number of jobs failed and finished every month during that time, and Figure 2.3 shows the failure rates. From September 2007, as the number of jobs doubled or even tripled, so did the failures. That is when PanDA was adopted by the ALTAS Collaboration. Subsequently, the failure rates were always over 20%.

Troubleshooters group different failures by their return codes, which we name it Failure type. Table 2.1 gives the top 5 failure types during the time period. By studying these failures, we find that although these failure types could separate different failures to some extent, they are far from enough for locating faults, since it is a common case that the failures of the same type have many different root causes. For example, when a job failed with pilotErrorCode 1099 (staging input file failed), the problem could be the pilot died, the input files were broken, the user’s permission was limited, and the file system was down and so on.

Table 2.1: Top 5 Failure types from 2006-10 to 2008-08

<table>
<thead>
<tr>
<th>Failure Type</th>
<th>Failed #</th>
<th>Fraction of Total</th>
<th>Failure Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskBufferErrorCode=100</td>
<td>697,032</td>
<td>16.14%</td>
<td>Job expired &amp; killed 3 days after submission (or killed by user)</td>
</tr>
<tr>
<td>jobDispatcherErrorCode=100</td>
<td>474,225</td>
<td>10.98%</td>
<td>Lost heartbeat</td>
</tr>
<tr>
<td>pilotErrorCode=1099</td>
<td>389,740</td>
<td>9.02%</td>
<td>Get error: Staging input file failed</td>
</tr>
<tr>
<td>taskBufferErrorCode=102</td>
<td>264,196</td>
<td>6.12%</td>
<td>Expired three days after submission</td>
</tr>
<tr>
<td>pilotErrorCode=1137</td>
<td>210,870</td>
<td>4.88%</td>
<td>Put error: Error in copying the file from job workdir to local SE</td>
</tr>
</tbody>
</table>

Figure 2.4: Number of production and analysis jobs under different attempts for all the jobs ending from 2007-10 to 2008-08
Figure 2.5: Success rates when attempt number $\leq x$ for all the jobs ending from 2007-10 to 2008-08

Figure 2.4 displays the distributions of both job successes and job failures under different attempts. Job with attempt $i$ means that the job is the $i^{th}$ submission of a task and all the previous submissions are failed. Jobs could be submitted up to 27 times, but here we only display the first 4 attempts since they have already shown the patterns. Here are 5 observations from the figure:

(1) A production job is resubmitted automatically if it fails on the previous attempt, since the number of job failures at attempt $i$ is almost the same as the sum of the numbers of job successes and job failures at attempt $i+1$. But analysis jobs were different: some jobs stopped trying after failing.

(2) The number of production jobs was about 4 times larger than the number of analysis jobs, which is computed based on the total number of finished and failed jobs that executed a $1^{st}$ attempt.

(3) Executing production jobs was more efficient than analysis jobs. The average numbers of attempts required for the success jobs are 1.28 and 1.40 for production and analysis. Also, the success rates for production jobs when attempt equaled 1, 2, 3, 4 were 81%, 64%, 47% and 49%, while the success rates for analysis jobs were 52%, 60%, 32% and 25%. It is clear to see that the success rates of production jobs were always larger than that of analysis jobs for the same number of attempts.

(4) Job is more likely to fail as the number of attempts increase. The proof has been listed in (3), since the success rate decreases with the increasing attempt number.
(5) Making more than four attempts is not very beneficial for both production and analysis jobs. In Figure 2.5, we can see that the success rate increases less than 1% when attempt number is larger than 3.

It tells us that production jobs dominate in PanDA, so they should be our major objects. Also the resubmitting could be stopped if the job has been submitted 3 times.

2.3 Similar jobs arrive in groups

Figure 2.6 displays the cumulative distribution functions of the interarrival time of creating and starting jobs in August 2008. Over 99% of jobs arrive within 1 second of the previous one, and the median of the interarrival times is less than 1 second. Job starts are not as close as job submissions, but still show strong temporal correlations, and the median is 1 second.

![Figure 2.6: CDF of Job creation interarrival time (left) and start interarrival time (right) in 2008-08](image)

In addition to the temporal correlation, job arrival in PanDA has a nice pattern: similar jobs arrive in groups. Group means that similar jobs coming continuously in a short time. We prove this by showing that the average size of the groups of similar jobs is larger than 40. As we discussed, task id is a good parameter to determine the similarity of production jobs, since the production jobs with the same task id (1) have the same code, (2) have input data stored on the same storage element (see Section 2.1).
In order to get the average group size, we first order all the jobs’ task ids by job creation time, and then use task ids to differentiate groups. Based on the data from 2008-04 to 2008-08, we found on average that 40 jobs with the same task id arrive continuously. Moreover, sometimes different users submit jobs at the same time, so that different groups might cross with each other, indicating some changes are not really the end of a group of similar jobs. Therefore, usually there are over 40 jobs with the same task id arriving together.

2.4 Temporal Correlation of failures

From Figure 2.7 and 2.8, we can learn that failure rate changes dramatically, and different types of failures have different behaviors. For instance, taskBufferErrorCode=100 failures come in burst – high rates in a short time, while the other two continue for a longer time with lower rates. Failure taskBufferErrorCode=100 occurs when the server kills the jobs as soon as it receives the kill requests from the users or the corresponding build job failures from the pilots. So this failure type shows very strong temporal correlation, and illustrates that users tend to kill a group of jobs together, or a group of pilots tend to report “build failed” at the same time.

Figure 2.9 demonstrates that all the failures are temporal correlated since all the distributions are heavy tailed. For all failure types, over 80% of the failure interarrival times are less than 1 minute, which is the result of three phenomena: (1) similar jobs arrive in groups; (2) similar jobs have the same I/O data block; and (3) schedulers tend to dispatch jobs to the pilots near the input data. Therefore, a group of similar jobs has high probability of sharing the same execution path in a short time. So if a system fault occurs, a number of similar failures might happen within a short time. That is to say, temporally correlated failures are likely to have the same root cause.

Particularly, 99.5% of taskBufferErrorCode=100 failures come less than 1 second after the previous one, a much stronger temporal correlation than the other two failure types. This matches the behaviors mentioned in the first paragraph.
Figure 2.7: Number of failures and jobs per hour in August 2008

Figure 2.8: Failure Rates of three failure types and all failures (bottom) in August 2008
Figure 2.9: CDF of Failure interarrival time of three failure types and all the failures (bottom right) in August 2008

2.5 Spatial Correlation

In addition to temporal correlation, most failures are also spatially correlated. Take a one-day record in Figure 2.10 for instance. Failures tend to fail on the same computing sites with short interarrival times. For example, all 1701 failures on site “MWT2_UC” only have 7 different task ids, and 74% of the failures share the same task id 24374. This observation is also anticipated by the three phenomena discussed in the previous Section.

We can also find this correlation from Figure 2.11: most failures were distributed on a small number of the computing sites. Furthermore, the number of failures on each site was not related to the total number of jobs on that site, which means most failures on the computing sites do not
occur because the sites have large capacities, rather because the jobs or the sites have problems.

Figure 2.10: Distribution of job failures on different sites every second in 2008-08-15

Figure 2.11: Distributions of failed and finished jobs on the top 60 computing sites ranked by the total number of jobs in 2008-08-15

Many other correlations exist between job failures and other metrics, we does not show them here. But all of them will play fundamental roles in the following learning mechanisms.
Chapter 3  
Clustering and Recognizing Job Failures

A large amount of logging, monitoring and testing information is generated every day in large-scale distributed system. Troubleshooters need to find the root causes of the failures from the data. In order to make this work more efficient, we use data mining and machine learning methods to extract and remember the patterns of failures. In this chapter, we will introduce previously proposed mechanisms in [7, 8, 19], and our extensions. The mechanisms (1) build classifiers to find the correlations between job metrics and job statuses in a sliding window; (2) extract signatures for every job based on the best correlation models; (3) cluster the signatures, and recognize similar failures by retrieving from labeled signatures. In the chapter, we begin with the workflow of the mechanism, and then describe the individual steps.

3.1 Overview of the mechanism

![Workflow of the learning mechanism](image)

Here are four important definitions for understanding the mechanism:

**Metric Vector (MV)** is a vector of metrics. The last attribute in MV is job status Class ∈ {c+, c⁻}, indicates job failure (c⁺) and job success (c⁻), or certain failure type (c⁺) and not the failure type(c⁻). We name the instance with Class = c⁺ positive sample, and the instance with Class = c⁻ negative sample. Other attributes are job metrics M, M = M₁ M₂ … Mₖ, where k = |M|. Each instance of MV is a training sample for a classifier, MV = <M, Class>.

**Ensemble of Models (EM)** is the collection of classifiers built and included this far. The
dataset for each classifier consists of the $MV$ instances in a training window (the sliding window at a certain time) [19]. Ideally, all the failures in the same window have the same root cause.

**Signature** is an indicator sequence $S = s_1 \ s_2 \ \ldots \ s_k$, where $s_i \in \{1, 0, -1\}$, $k = |M|$. $s_i = 1$ if the value of $i^{th}$ metric $m_i$ is abnormal; $s_i = -1$ if $m_i$ is normal; $s_i = 0$ if $m_i$ is irrelevant [8]. The definitions of abnormal, normal and irrelevant will be stated in Section 3.5.

**Cluster** is a group of similar signatures. Ideally, all the signatures in one cluster have the same root cause of failure, or state of success [8].

Figure 3.1 illustrates the workflow of the mechanism. On receiving an instance of metric vector, the mechanism first checks whether the training window contains the minimal numbers of instances of every class; if yes, it builds a classifier, and updates the ensemble of models if the new classifier is good enough. Then the mechanism chooses the best model from the ensemble, and extracts a signature for the instance. When a number of signatures are generated, clusters these signatures in order to help labeling all the instances. Finally, it retrieves all the existing labeled signatures and outputs the most likely label of the instance.

### 3.2 Assumption of building classifier

We assume that in most of the cases, most failures inside one training window are caused by the same fault. If the assumption stands, most failures in the same window will behave similarly. Then the classifier would be meaningful in indicating a subset of relevant metrics for a particular fault.

Fortunately, the assumption is true in PanDA. As we saw in Section 2.4, job failures in PanDA have strong temporal correlations, and temporally correlated failures are likely to have the same root cause. Therefore it shows the great possibility that the number of failures caused by a same fault will be much larger than that of the other failures in the window, which is usually short.

Besides, our experimental results for learning sliding window size in Section 4.2 also support the assumption. The high mean and low standard deviation of classification accuracy illustrate that the classifiers indeed find the accurate correlations between metrics and statuses in most windows, and successfully use them to predict job statuses in the near future.
3.3 Building correlations between job metrics and job statuses

Once the training window is full, we learn a Bayesian network classifier to classify job failures and successes. The building process is a greedy algorithm, selecting the metric set on which the most accurate classifier is built [7]. The pseudocode is as follows:

**Build Classifier** (Training Window)

Initialize Metric Set (MS) to empty, set MAX-BA to zero

While MAX-BA is not high enough and MS is not too large then

For each metric E that is not in MS do

Train Naïve Bayesian Network Classifier only on the metrics in MS and E

Compute Balanced Accuracy (see definition below) on the classifier

End-For

Set MAX-BA to the highest Balanced Accuracy in the above loop

Choose the metrics generating MAX-BA, and add them to MS

End-While

**Balanced Accuracy (BA)** is defined as the average value of the probability of correctly classifying the positive samples and the probability of correctly classifying the negative samples [7]:

\[ BA = \frac{P(F(\bar{M}) = c^+|Class = c^+) + P(F(\bar{M}) = c^-|Class = c^-)}{2} \]  

(1)

\( F \) represents the classifier. If and only if both the probabilities are high, balanced accuracy will be high, no matter what the ratio between the two classes of samples is.

Bayesian networks are chosen here because they are easy to interpret and modify [14, 7]. The conditional probability table clearly shows the contribution of a certain value of a metric to the classification, which plays a crucial role in finding correlations among variables and classes. On the other hand, a network can be modified by experts easily according to their experiences.

Our mechanism above has two extensions to the method proposed by Cohen et al [7]. We found that when choosing the metrics that generate MAX-BA each step, many metrics are equivalently good in PanDA. Then two possible choices could be made: select all the equivalent
metrics, or select only one of them. The algorithm in [7] did not discuss the problem. In this paper, we use the previous one since it preserves more correlation information (between metrics and job statuses) than the other. The comparison experiment in Section 4.4 confirms the choice.

The other one is that we use Naïve Bayesian networks instead of Tree-Augmented Bayesian networks (TAN). While both networks yield almost the same accuracies, the naïve one saves much time (shown in Section 4.4).

3.4 Updating ensemble of models

In complex systems, various faults occur expectedly or unexpectedly. And different faults have different correlations to the metrics. Therefore one single classifier to capture all the correlations is obviously not good enough [19]. Learning a dynamic model in a sliding window (i.e. update the classifier on recent samples) could make the classifier up to date, but it has main disadvantages. One is that it has to relearn the model again even a similar situation has happened in the past [19]. The other very important disadvantage is that we might lose many good models learned before, because sometimes failures with very different root causes appear in the same window, reducing the accuracy of the current classifier, but the better classifiers are not remembered.

Thus we use an ensemble of models to store all the good models. Once a training window is full, build a Bayesian network classifier for the window as described in the previous Section. If the balanced accuracy is higher than all older classifiers or enough high, add it into the ensemble. After that, extract a signature for each sample (see the next section), no matter failure or success.

Our extension to the algorithm proposed in [19] is first separating the dataset by failure types, and then building models for each failure type independently. Since the classifier builds correlation between metrics and statuses in a short window, then if most failures in the window are caused by the same fault, this correlation is more likely to indicate the features of that fault by the metrics. So we want one kind of fault dominate the window as much as possible. And failure type might help separating failures of different faults into different windows, because in a short time, failures with the same failure type are more likely to share the same fault than the one with other failure types. The experiment gives positive result to this extension as shown in Section 4.2.

The pseudocode is the following [19]:
**Update Ensemble of Models** (Dataset, minimum number of samples per class)

For each Failure Type dataset do

Initialize Ensemble of Models (EM) to empty, and Training Window (TW) to empty

For each new sample do

Add the sample to TW

If TW has minimum number of samples per class then

Build Classifier Model M on current TW as in Section 3.3

Compute Balanced Accuracy on M

If M has higher BA than other models in EM, or BA is high enough then

Add M to EM, and set TW to empty

End-If

End-If

Extract Signature on the sample, EM, and past window D (See Section 3.5)

End-For

End-For

Another question here is how to choose the size of the training window for each failure type. Let \((P, N)\) denote window size, where \(P\) denotes the number of positive samples and \(N\) for the number of negative samples. There is a tradeoff between model generalization and accuracy. If the window is too small, the models might not be generalized to new samples due to overfitting. But if too many samples are included in the window, the possibility of mixing more than one kind of fault increases and the accuracy of the models will decrease. Besides, the ratio between \(P\) and \(N\) influences the learning result too. So we decided to find the window size experimentally [5, 19]: varying the size of the training window, learn a dynamic model (an updating model which rebuilds the classifier once the window is full) based on that window size, and choose the best size that yields the highest average balanced accuracy.

In the work, we have not considered removing the models from the ensemble. Two reasons for that: one is storing a classifier is very cheap both in time and in memory [19]; the other is that the number of good models is not large, less than 250 in 15 days (see experiment result in Section 4.2). If the size of the ensemble grows too large, we could use Least Frequent Used policy etc.
3.5 Extracting signatures

After updating the ensemble, signature is extracted to represent the most confident correlation for each sample based on the best model from the ensemble at that time.

The best model $F_B$ should be the one generates the most accurate and confident classification given the metric values. Brier score computes the overall squared error between the classification belief and the true class for all the samples in a short window of past data $D$ [8]:

$$BS_{F_j}(D) = \sum_{k=1}^{[D]} [P\{F_j(M_k) = c^+\} - I(Class_k = c^+)]^2$$  \hspace{1cm} (3)

where $D$ is the set of latest samples whose size is the same as the training window, and $F_j$ is the $j^{th}$ model in the ensemble. Function $I$ equals 1 if the instance is positive, and 0 otherwise. Probability $P$ is the conditional probability of the instance being positive given the value of $M$. The lower the score is, the more believable and accurate the model is. So $F_B$ should be the model with the lowest Brier score.

For a given instance in a given model, if

$$P(M_i = m_i|Class = c^+) \geq P(M_i = m_i|Class = c^-)$$  \hspace{1cm} (2)

We say $m_i$ (the value of $M_i$) is abnormal, since $m_i$ appears more often when the instances are positive than they are negative. If the inequality (1) reverses, $m_i$ is normal. If $M_i$ is not selected in $F_B$, $m_i$ is irrelevant. In this way, we can assign the strengths of correlations to all the metrics. The pseudocode for extracting signatures is the following [8]:

**Extract Signature** (a sample, EM, D)

Compute the brier score on $D$ for all the models in EM, and find $F_B$

For each metric $M_i$ in sample do

If the value $m_i$ is not selected in $F_B$, then $S_i = 0$

Else if $m_i$ satisfies inequality (1) in $F_B$, then $S_i = 1$

Else $S_i = -1$

End-If

End-for
3.6 Clustering signatures

Because of noise in the dataset and some biased windows, it is possible that the signatures of the same fault are not the same for different failures. Noise is the ambiguous, inconsistent, and incomplete data, and biased window means that the samples of one class in the window are generated under a particular situation not related to the root cause. For example, all the failed jobs in D have the root cause - authentication problem, but happen to have the same task id A, then the classifier having task id is likely to be selected as FB under this particular situation, and task id A will be marked abnormal in the signatures. But actually, task id has nothing to do with the real root cause.

Clustering signatures aims to group the similar signatures together to capture their common characteristics better and reduce the above two problems to some extent. Usually, the centroid of a cluster shows most of these common characteristics. The generated clusters can help experts labeling the signatures in those clusters.

We chose K-Medoids as our clustering algorithm instead of K-Means for these two reasons: (1) the instances of our clustering data are indicators (-1, 0, 1), so the numeric average value does not make much sense; (2) the centroids in K-Medoids are more meaningful as cluster representatives. We use Euclidean Distance to measure the distance between two signatures.

To measure the purity of clusters, we compute the entropy [8] of each cluster:

\[
H = - \sum_{i=1}^{n} p_i \log_{n}(p_i) , \quad \sum_{i=1}^{n} p_i = 1, n \geq 2
\]

(4)

where n is the total number of labels, and \( p_i \) is the percentage of the samples with \( i \)th label in the cluster. The purist cluster is generated when all the samples in that cluster have the same label, \( H \) is defined as 0. But when the worst case happens, i.e. every label has \( 1/n \) of the samples, \( H \) has value 1. So the smaller the entropy is, the purer the cluster.

In the end, troubleshooters and experts can look at the clusters with low entropies, identify the root causes of them, and then label these signatures in the clusters.

3.7 Retrieving Signatures

When a new signature is extracted, finding the label that the signature most likely to have is
the most interesting thing and the output of our mechanism.

First, the N closest signatures are retrieved from all the existing labeled signatures, and then the plurality of the labels attached to these N neighbors is chosen as the label of the new signature [8]. In this way, people can recognize the failures happened before by signature retrieval.
Chapter 4
Experiments and Results

All the experiments were run on the real job records from the PanDA server database. Each job record contains 29 metrics, including one class metric – failure type. These metrics include various job execution information, such as scheduler id, task id, computing site, input data size, and so on. Please see the table in appendix for details. We evaluate the correctness of the ensemble of models, the effectiveness of the signature clustering on the top three failure types, and the performance comparison before/after our extensions of the mechanisms.


4.1 Learning the training window size

As shown in Section 2.4, different failure types have different temporal correlations. Some occur discontinuously but in high-volume, such as taskBufferErrorCode=100, while some happen more continuously and in smaller quantity, e.g. jobDispatcherErrorCode=100. So learning training window sizes for different failure types is important.

We set the window size by choosing the best area in a learning surface. The learning surface is built as follows: vary the number of positive and negative samples independently, and for each configuration, learn a dynamic classifier on a same dataset and evaluate the window size with the overall balanced accuracy. In the experiment, the training window size increases from (10, 10) to (50, 100), and the dataset is five-day job history from 2008-08-11 to 2008-08-15.

Figure 4.1 shows the learning surfaces for three failure types. The darker the area is, the higher the accuracy. Failure type taskBufferErrorCode=100 prefers smaller positive samples, which is in line with its bursty nature. And in general, positive samples should be fewer than negative samples, and both should not be larger than 70. Thus by maximizing window size under the condition of high accuracy, the decisions for the three failure types are listed in Table 4.2.
Figure 4.1: Overall balanced accuracy in 5-day records with different sliding window size of three failure types
4.2 Prediction performance

To evaluate the correctness of the ensemble of models, we predict failures on 15-day job history from 2008-08-01 to 2008-08-15. Table 4.1 gives the size of the datasets of three failure types. After initialization (building the first classifier), the samples in the next training window are classified by the model with the highest BA in the current ensemble. The histograms of BAs of all the windows for different failure types in Figure 4.2 show that the models have very high BA most of the time. Also from Table 4.2, we can see that the balanced accuracies of three failure types all have high means and low standard deviations; high true positive rates (also called hit rate) indicate that over 94% of the positive samples are correctly identified beforehand; low false positive rates (also called false alarm rate) show that less than 3% of samples are wrongly classified as the failure type; and brier scores are lower than 1 most of time, which says that the models learned are indeed both correct and confident in their classifications. All these measures are stating that the ensemble of models can really capture the correlations between metrics and statuses for most of the samples.

Figure 4.2: Histograms of BA of all windows when learning on 15-day data for three failure types
Table 4.1: Sizes of the datasets of three failure types in 15-day job history

<table>
<thead>
<tr>
<th></th>
<th>taskBufferErrorCode=100</th>
<th>jobDispatcherErrorCode=100</th>
<th>pilotErrorCode=1099</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Samples</td>
<td>481,089</td>
<td>481,089</td>
<td>481,089</td>
</tr>
<tr>
<td># of Failures</td>
<td>3,625</td>
<td>11,940</td>
<td>27,549</td>
</tr>
</tbody>
</table>

Table 4.2: Test performance of three failure types in 15-day job history

<table>
<thead>
<tr>
<th></th>
<th>taskBufferErrorCode=100</th>
<th>jobDispatcherErrorCode=100</th>
<th>pilotErrorCode=1099</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window Size</td>
<td>12, 40</td>
<td>25, 50</td>
<td>25, 40</td>
</tr>
<tr>
<td>Balanced Accuracy (%)</td>
<td>96.04 ± 9.85</td>
<td>98.94 ± 4.22</td>
<td>98.43 ± 4.91</td>
</tr>
<tr>
<td>True Positive Rate (%)</td>
<td>94.19 ± 18.64</td>
<td>98.69 ± 7.68</td>
<td>98.45 ± 7.61</td>
</tr>
<tr>
<td>False Positive Rate (%)</td>
<td>2.11 ± 5.74</td>
<td>0.81 ± 2.87</td>
<td>1.50 ± 3.30</td>
</tr>
<tr>
<td>Brier Score</td>
<td>0.67 ± 0.37</td>
<td>0.87 ± 0.25</td>
<td>0.90 ± 0.18</td>
</tr>
<tr>
<td># of new models generated</td>
<td>49</td>
<td>441</td>
<td>1037</td>
</tr>
<tr>
<td># models in final ensemble</td>
<td>38</td>
<td>52</td>
<td>143</td>
</tr>
<tr>
<td># of times using new model</td>
<td>3</td>
<td>3</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 4.2 also shows the robustness and generalization of the ensemble of models. We found that only a small number of models are included in the final ensembles, compared to the total number of the new models generated. Moreover, when using the most accurate model to classify the samples in the next window, old models are selected instead of the new ones most of the time. This result illustrates that models learned are indeed generalized and robust, since the right models saved in the ensemble can correctly classify the new samples.

4.3 Clustering evaluation

As described in Section 3.6, we use entropy to evaluate clusters. The smaller the entropy is, the better the cluster. Figure 4.3 gives the average entropy when clustering with different number of clusters. We can see that for three failure types, the average entropies are between 0.1 and 0.3. When the total number of clusters (K in K-Medoids) is larger than 11, the entropies of the three failure types do not decrease much, which means that increasing K cannot generate purer clusters any further.

Two possible reasons for the impure clusters: (1) Insufficient metrics. The metrics are not enough to characterize some faults, so the current models cannot differentiate the failures by the
signatures; (2) Signatures are extracted based on very biased windows, so that quite different signatures are generated, compared with unbiased windows. Therefore, these incorrect signatures increase the entropy of the clusters.

Table 4.3 lists one positive and one negative cluster for the first two failure types, and two positive clusters for the last one. First of all, we could see that most of 1 are in positive clusters, and -1 are in negative clusters, indicating that the values are indeed abnormal (value 1) in the job failures and normal (value -1) if otherwise. Also, these four positive clusters use different metrics, so that they can be differentiated from each other. And those metrics are meaningful. For instance, jobDispatcherErrorCode = 100 (lose heartbeat) is the only failure type which chooses modificationTime (the value changes each time the job status changes) as abnormal in positive cluster. The server checks all the jobs each hour and kills the ones whose modification times are 6 hours ago. So the failure type in one window should have similar modification times. Another example is inputSize, which was chosen as abnormal in both positive clusters of pilotErrorCode=1099 (stage input files error), but not for the other two failure types. We find that the values of inputSize of the failures in the two clusters are always zero. Then it is clear to explain why “no input data transferred” is one indicator for “stage input file error”. Also we can further prove the spatial correlation of all the failures, since computingSite has been selected as abnormal for all the three failure types.
Table 4.3: Selected metrics in cluster centroids of three failure types in 15-day job history; % Pos and %Neg in the title show the percentages of positive and negative samples in that cluster

<table>
<thead>
<tr>
<th>Metric</th>
<th>taskBufferErrorCode=100</th>
<th>jobDispatcherError=100</th>
<th>pilotErrorCode=1099</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>96.62% Pos</td>
<td>100% Neg</td>
<td>95.14% Pos</td>
</tr>
<tr>
<td>schedulerID</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>prodUserID</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>taskID</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>creationTime</td>
<td>0</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>startTime</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>modificationTime</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>modificationHost</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>computingSite</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>transformation</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>homepackage</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prodSourceLabel</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cpuConsumptionUnit</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>ipConnectivity</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>attemptNr</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>assignedPriority</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>currentPriority</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>relocationFlag</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prodDBlock</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>dispatchDBlock</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>destinationDBlock</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>cloud</td>
<td>0</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>timeGetJob</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>timeStageIn</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>timeExe</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>timeStageOut</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>inputSize</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>inputStatus</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>destinationSE</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
</tbody>
</table>

Also we can observe several kinds of faults from the same failure type. The two positive clusters in pilotErrorCode=1099 have different values in startTime, homepackage, and timeStageIn. The first cluster marks all of them abnormal, while the second cluster marks them irrelevant. startTime is the time when the pilot starts staging input files from local storage element.
to local worker node, *homepackage* is the version of ATLAS software (includes a name and four-digit version number), and *timeStageIn* is the time consumed for staging inputs. We could infer that these clusters should be caused by the different faults. The failures in the first cluster might because the right software has not been found so that the input data transferring has not been started yet; as a result, *startTime* and *timeStageIn* are all zeros of the failures in the cluster. And the failures in the second cluster might be input data missing, so that the data transferring connection has been built (so *startTime* and *timeStageIn* are not zeros) but nothing was transferred before exiting.

We can see that, the generated clusters can indeed differentiate or indicate some meaningful root causes of failures, even with only 29 metrics.

### 4.4 Performance without extensions

In Section 3.4 and 3.3, we described the three extensions in our mechanisms: (1) first separate dataset by failure types, then build models for each failure type independently; (2) when building classifier, select all the equivalent metrics that yield the largest accuracy incensement each step; (3) use Naïve Bayesian network instead of TAN. Although they are small extensions, their effects are good.

We run the mechanism on the dataset that contains all three failure types, in order to see the differences with and without the first extension. The sliding window size chosen from the learning surface is (15, 50). “All failures” in Figure 4.3 shows the cluster entropy in the setting. We can see that in almost all the cases, the entropy is higher than that of any separate failure type, illustrating that separating dataset by failure types can improve the average purity of the groupings. We also compute the entropy using four labels (three different failure types, and otherwise). The entropy is around 0.32 when K is 15, which is a promising result of separating different failures in different groups when the labels are very vague.

Figure 4.4 shows the comparison of cluster entropy between two methods in building classifier: selecting all the equivalent metrics that yield the largest accuracy incensement each step or only one of them. We can see that the former produces lower cluster entropies for all three failure types, which means the groupings are purer in our approach.
In the end, we test two networks, Naive Bayesian network and TAN, on the same dataset with 29 metrics and 12025 samples, with the same training window size (15, 50). From table 4.4, it is easy to see that Naive Bayesian Network took less than 1/6 of the time used by TAN. And Naive Bayesian Network only took less than 0.3 second to find the best classifier on a training window.

Table 4.4: Time costs when building classifiers using Naive Bayesian Network and TAN

<table>
<thead>
<tr>
<th>Time</th>
<th>Naive Bayesian Network</th>
<th>TAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build a classifier (ms)</td>
<td>9.33</td>
<td>56.52</td>
</tr>
<tr>
<td>Average number of classifiers generated per step</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>Extract a signature (ms)</td>
<td>0.019</td>
<td>0.034</td>
</tr>
</tbody>
</table>

Figure 4.4: Average cluster entropies of two approaches to build classifiers
Chapter 5

Related work

Diagnosing failures using data mining and machine learning methods is of growing interest because of the widespread use of distributed systems. Countless failures happen, along with a huge amount of logging and monitoring information, making them fine environments for this kind of work. Many research labs and universities have started using various methods to discover the patterns of failures [1, 2, 6], or diagnose failures automatically.

Li et al. [6] show some interesting characteristics of job failures in LHC Computing Grid. Temporal and spatial burstiness of job failures are observed and job life spans exhibit strong autocorrelations. In this paper, we also find that job failures are temporally and spatially correlated in PanDA. And we strongly suspect that the correlations are caused by the pattern of job arrival: similar jobs arrive in groups. Temporal relationship among events is a basic premise of failure diagnosis [13]. Fu et al. [18] build models to quantify the temporal and spatial correlations in coalitions of clusters, and use these models for failure prediction.

There are mainly three kinds of techniques for fault localization [13]. The first one is expert-system techniques, such as rule-based and model-based representations of expert experiences. But these models may be difficult to obtain and keep up to date.

The second kind is model traversing techniques, which detect failed system components by using the representations of relationships among network entities. Sherlock from Microsoft [15] constructs an inference graph based on packet traces, to represent the dependencies among all the components in an enterprise network among three layers; based on the inference graph and measurements of service response time, the system then uses a probabilistic correlation algorithm to produce a list of suspect components ranked by their likelihood of being the root cause. Pinpoint [12] records all the components used in every client request, and also the believed status (success or failure) of the request. Then it chooses the most correlated components to the failures by data clustering and statistical techniques.

The last one is graph-theoretic techniques, including divide and conquer algorithm, codebook, context-free grammar, and belief network [13]. These techniques rely on fault propagation model
(FPM), a graphical model of the system. The model includes the representation of all faults and symptoms that occur in the system, and builds causal or dependency relationships among them. Then the fault localization algorithm analyzes the FPM to identify the best explanation of the observed symptoms. Intelligent probing developed in IBM [10] is an example of both codebook and belief network. A probe is a command or transaction involving some network elements, and only when all the related elements are OK, the probe succeeds. So the probes are symptoms and the network elements are faulty components. Given an initial probe set and a dependency matrix of these probes, intelligent probing finds near-minimal probe set in linear time, and uses a Bayesian network approach and a local-inference approximation scheme to determine the most likely configuration of the states of the network elements. Furthermore, they extended the work to active probing [9] and reduced the number of probes needed dramatically. However for large-scale distributed systems, building fixed structures of network elements is not quite practical.

For large-scale distributed systems, researchers tend to build the fault propagation models on logging and monitoring data using data mining and machine learning techniques. Cielask et al. [4] constructs a decision tree on the job and resource labels, to select the most significant decision points for some failures in grids. Hellinger Distance Decision Tree algorithm is applied to improve the classification performance for unbalanced data. The work shows some interesting results, and some correlations that suggest the root causes.

The mechanisms used in this paper [7, 8, 19] belong to the last kind. It generates the belief networks of correlations between system metrics and states, assuming that only one fault exists in the system at a time. The symptoms in this case are metrics, and FPM is the correlation model among metrics and states. After generating the models, the mechanisms use a statistical method to extract signatures for each sample, in order to recognize the similar problems in the future.
Chapter 6

Future work

There are four possible extensions to improve the current mechanism.

The first one is very practical: include more metrics. As we can see, 29 metrics are far from enough if we want to touch deeper causes. Then the questions are how to get the useful metrics and how to deal with a high dimensional dataset. For the first question, we may collect the key metrics of all the components along the job execution trace. If we have too many metrics, in order to overcome the increased noise and curse of dimensionality, more samples are needed in the training window to maintain accuracy; but if more samples are included, the possibility of having more than one kind of root cause increases, and the accuracy of model decreases. For the second question, multi-source temporal segmentation could be a solution [3]: instead of learning all the metrics together, the technique builds independent models on multiple components in parallel, and notifies other data sources once a model indentifies the fault. Dimension reduction on metrics can also be helpful. We found that some metrics are closely correlated with each other no matter what the job status. For example, destinationDBlock or prodDBlock always have the same value if taskID is fixed. Since large metric set causes trouble, it would be useful to reduce the dimension before learning. Of course, expert knowledge can be introduced before dimension reduction.

Continual development is the nature of large-scale distributed systems, so our mechanism should be as well. But the current clustering algorithm cannot update clusters dynamically. At the same time, clustering a large number of samples together is extremely time-consuming. In the next step, we need to adopt incremental clustering to overcome these problems.

Another extension is extracting more uniform signatures no matter the windows are biased or not. As mentioned in Section 4.3, very biased windows can generate quite different signatures, and that is because the samples of one class in the window are generated under a particular situation not related to the root cause. Then if we include some representative samples in various situations, biased window might be less biased or even unbiased. But the problem is that the representative samples are easy to be negative samples [24], but hard to be positive samples, since it might not be a good idea to include different failures in one training window if we are not sure about their root
causes.

Last but not least, we should discuss the most likely labels for all the signatures in the pure clusters with experienced troubleshooters, label these signatures, and evaluate the effectiveness of failure retrieval.
Chapter 7

Conclusions

Due to the diversity and development nature of large-scale distributed systems, job failures are inevitable and extremely difficult to diagnose. This paper works on the problem by learning correlations between job metrics and job statuses, and extracting signature based on the most confident correlation model for every job. Naïve Bayesian network classifier is used for building these models. Then clustering all the signatures captures the common characteristics of different failures, helping troubleshooters to analyze and label the failures with their root causes. Recurring problem can be recognized by searching the failure’s closest labeled signatures. Moreover, new models are learned continuously in a sliding window, and all the good ones are remembered in an ensemble of models in order to update the system.

We implemented the mechanism on a typical large-scale distributed system – PanDA, and run experiments on 15-day job records. The great classification performances show that most models are indeed both accurate and confident in classification. And the centroid signatures of the clusters can indeed differentiate or indicate some meaningful root causes of the failures, even with only 29 metrics. So we consider the mechanism to be a promising approach for automatically diagnosing job failures.

In addition, the paper also showed that job failures in PanDA are both temporally and spatially correlated, which provides a solid foundation for our mechanism. And we believe that the jobs arrival pattern - similar jobs arrive in groups - is likely to be the reason for these correlations.
References


[26] https://twiki.cern.ch/twiki/bin/view/Atlas/Panda
[27] http://www.opensciencegrid.org/
[29] http://rapid-i.com/
[31] http://atlas.ch/
# Appendix

Table A.1: Metrics selected in training samples from PanDA archive database

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schedulerID</td>
<td>ID created by scheduler</td>
</tr>
<tr>
<td>prodUserID</td>
<td>ID of the user defined the job (user's certificateDN or e-mail address)</td>
</tr>
<tr>
<td>taskID</td>
<td>task ID</td>
</tr>
<tr>
<td>creationTime</td>
<td>generated by MySQL function (client side)</td>
</tr>
<tr>
<td>startTime</td>
<td>will be filled after job finishes</td>
</tr>
<tr>
<td>modificationTime</td>
<td>generated by MySQL function (client)</td>
</tr>
<tr>
<td>modificationHost</td>
<td>can be extracted from MySQL function (client)</td>
</tr>
<tr>
<td>computingSite</td>
<td>example: BU_ATLAS_Tier2</td>
</tr>
<tr>
<td>transformation</td>
<td>example: share/rome.1001.reco.MuonDigit.OverrideRE.trf</td>
</tr>
<tr>
<td>homepackage</td>
<td>example: JobTransforms-10.0.1.5</td>
</tr>
<tr>
<td>prodSourceLabel</td>
<td>possible values: &quot;managed, regional, user, panda, test&quot;</td>
</tr>
<tr>
<td>cpuConsumptionUnit</td>
<td>will be filled after job finishes</td>
</tr>
<tr>
<td>ipConnectivity</td>
<td>comes from CERN prodDB for managed production jobs</td>
</tr>
<tr>
<td>attemptNr</td>
<td>number of job-attempt, (&quot;0&quot; in jobsDefined table for new jobs,&quot;1&quot; when job is moved to jobsActive table)</td>
</tr>
<tr>
<td>cpuConsumptionTime</td>
<td>will be filled after job finishes</td>
</tr>
<tr>
<td>assignedPriority</td>
<td>comes from CERN prodDB for managed production jobs (assigned by production manager)</td>
</tr>
<tr>
<td>currentPriority</td>
<td>indexed, assigned by production manager, usually the same as assignedPriority</td>
</tr>
<tr>
<td>relocationFlag</td>
<td>flag for submitting special jobs (M5reco) to a given cluster</td>
</tr>
<tr>
<td>prodDBlock</td>
<td>name of datablock where job's input files(s) is part of</td>
</tr>
<tr>
<td>dispatchDBlock</td>
<td>name of job's dispatch datablock; a prodDBlock may be broken down into smaller blocks for dispatch to sites</td>
</tr>
<tr>
<td>destinationDBlock</td>
<td>name of job's destination datablock; is used to register the outputs of an associated set of jobs as belonging to one block to be saved at an archival destination, index</td>
</tr>
<tr>
<td>cloud</td>
<td>Cloud (associated with Tier 1) job is submitted to (US,Canada,etc.)</td>
</tr>
<tr>
<td>inputSize</td>
<td>The total size of input files, computed from filesTable, in bytes</td>
</tr>
<tr>
<td>inputStatus</td>
<td>The status of input files; whether job's inputs are at computingSite, job's output is available in destination datablock, index</td>
</tr>
<tr>
<td>destinationSE</td>
<td>destination storage element (archival destination) of job's output file(s)</td>
</tr>
<tr>
<td>timeGetJob</td>
<td>time for pilot to fetch the job</td>
</tr>
<tr>
<td>timeStageIn</td>
<td>time for pilot to stage the input files to the worker node</td>
</tr>
<tr>
<td>timeExec</td>
<td>time for pilot to execute the job</td>
</tr>
<tr>
<td>timeStageOut</td>
<td>time for pilot to stage the output files to local storage element</td>
</tr>
<tr>
<td>failureType</td>
<td>a particular failure type</td>
</tr>
</tbody>
</table>