Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis

Qiang FU[1], Jian-Guang LOU[1], Yi WANG[2], Jiang LI[1]

[1]Microsoft Research Asia
Beijing, P.R. China
{qifu, jlou, jiangli}@microsoft.com

[2]Dept. of Computer Science & Technology
Beijing University of Posts and Telecommunications
Beijing, P.R. China
wangyi.tseg@gmail.com

Abstract -- **Detection of execution anomalies is very important for the maintenance, development, and performance refinement of large scale distributed systems. Execution anomalies include both work flow errors and low performance problems. People often use system logs produced by distributed systems for troubleshooting and problem diagnosis. However, manually inspecting system logs to detect anomalies is unfeasible due to the increasing scale and complexity of distributed systems. Therefore, there is a great demand for automatic anomaly detection techniques based on log analysis. In this paper, we propose an unstructured log analysis technique for anomaly detection. In the technique, we propose a novel algorithm to convert free form text messages in log files to log keys without heavily relying on application specific knowledge. The log keys correspond to the log-print statements in the source code which can provide cues of system execution behavior. After converting log messages to log keys, we learn a Finite State Automaton (FSA) from training log sequences to present the normal work flow for each system component. At the same time, a performance measurement model is learned to characterize the normal execution performance based on the log messages' timing information. With these learned models, we can automatically detect anomalies in newly input log files. Experiments on Hadoop and SILK show that the technique can effectively detect running anomalies.**

Keywords - log analysis; distributed system; problem diagnosis; FSA

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, 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 execution anomaly monitoring and detection tools becomes an essential requirement of many distributed systems to ensure the Quality of Service.

There are two classes of typical anomalies: one is work flow errors - errors occurring during the execution paths; the other is execution low performance - the execution time takes much longer than normal cases although its execution path is correct. In this paper, we present an unstructured log analysis technique that can automatically detect system anomalies using commonly available system logs. It requires neither additional system source code instrumentation nor any runtime code profiling. The technique mainly consists of two processes: the learning process and the detection process. The goal of the learning process is to obtain models that represent the normal execution behavior of the system from those logs produced by normally completed jobs. The input data for the learning process is training log files printed by different machines. At first, we convert the log message sequences in the log files into log key sequences. Log keys are obtained by abstracting log messages. Then, we a derive Finite State Automaton (FSA) to model the execution path of the system. With the learned FSAs, we can identify the corresponding state sequences from training log sequences. Next, we count the execution time of each state transition in state sequences, and obtain a performance measurement model through statistical analysis. In the detection process, for newly input log sequences, we check them with those learned models to automatically detect anomalies. It should be noticed that the system's normal behavior may change after an upgrade. Therefore, it is necessary to re-train the model after each system upgrade.

Assumptions: In our technique, system anomaly detection is based on the cues gained from the previous normally completed jobs' 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.

The paper is organized as follows. In section 2, several related research efforts are briefly surveyed. The log key extraction and FSA construction are introduced in section 3 and section 4. In section 5, we discuss the performance measurement model construction. After that, anomaly detection is described in section 6. Then, experimental results are presented in section 7. Finally, section 8 concludes the paper.

II. RELATED WORK

Monitoring and maintaining techniques that make use of execution logs are the least invasive and most applicable, because execution logs are often available during a system's daily running. Therefore, analyzing logs for problem diagnosis has been an active research area for several decades. In this paper, we only survey the approaches that perform the analysis automatically.

One set of algorithms [1, 2, 3, 4] judge the job's trace sequence as a whole, where a log sequence is often simply recognized as a symbol string. Dickenson et al [1] collect execution profiles from program runs, and use classification techniques to categorize the collected profiles based on some string distance metrics. Then, an analyst examines the profiles of each class to determine whether or not the class represents an anomaly. Mirgorodskiy et al [2] also use string distance metrics to categorize function-level traces, and identify outlier traces or anomalies that substantially differ from the others. Yuan et al [4] propose a supervised classification algorithm to classify system call traces based on the similarity to the traces of known problems. In other literature, a quantitative feature is extracted from each log sequence for error detection. For example, in [3], the authors preprocess the logs to extract the number of log occurrence times as a log feature, and detect anomalies using principal component analysis (PCA). These kinds of algorithms can find whether the job is abnormal, while can hardly obtain the insight and accurate information about abnormal jobs.

Another set of algorithms [5-8] view system logs as a series of footprints of systems' execution. They try to learn FSA models from the traces to model the system behavior. In the work of Cotroneo et al [5], FSA models are first derived from the traces of Java Virtual Machine collected by the JVMMon tool [6]. Then, logs of unsuccessful workloads are compared with the inferred FSA models to detect anomalous log sequences. SALSA [7] examines Hadoop logs to construct FSA models of the Datanode module and TaskTracker module. In [8], based on the traces that record the sequences of components traversed in a system in response to a user request, the authors construct varied-length n-grams and a FSA to characterize the normal system behavior. A new trace is compared against the learned FSA to detect whether it is abnormal. In their algorithm, a varied-length n-gram represents a state of the FSA. Unlike these methods, which heavily depend on application specific knowledge including some predefined log tokens and the stage structure of Map-Reduce, our algorithm can work in a black-box style. In addition, our algorithm is the only one that uses timing information in the log sequence to detect the low performance problem.

In some other literature [17, 18], logs are used to perform troubleshooting related tasks in different scenarios. GMS [17] detects abnormal machines with wrong configurations. It extracts features from the data source and applies the distributed HilOut algorithm to identify the outliers as the misconfigured machines. Its data source includes log files, utility statistics and configuration files. In [18], a decision tree is learned to identify the causes of detected failures where the failures have been detected beforehand. It records the runtime properties of each request in a multi-tier Web server, and applies statistical learning techniques to identify the causes of failures. Unlike them, our algorithm mainly tries to detect anomalies through exploiting the timing and circulation information.

III. LOG KEY EXTRACTION

Systems logs usually record run-time program behaviors, including events, states and inter-component interactions. An unstructured log message often contains two types of information: one type is free-form text string that is used to describe the semantic meaning of a recorded program behavior; the other type is a parameter that is used to express some important system attributes. In general, the number of different log message types is often huge or even infinite because of various parameter values. Therefore, during log data mining, directly considering log messages as a whole may lead to the curse of dimension.

In order to overcome this problem, we replace each log message by its corresponding log key to perform analysis. The log key is defined as the common content of all log messages which are printed by the same log-print statement in the source code. In other words, a log key equals to the free-form text string of the log-print statement without any parameters. For example, the log key of log message 5 (shown in Figure 1) is "***Image file of size saved in seconds***". We analyze logs based on log keys because: (1) In general cases, different log-print statements often output different log text messages. It means that each type of log key corresponds to one specific log-print statement in the source code. Therefore, a sequence of log keys can reveal the execution path of the program. (2) The number of log key types is

finite and is much less than the number of log message types. It can help us to avoid the curse of dimension during data mining.

The challenging problem is that we know neither which log messages are printed by the same log-print statement nor where parameters are in log messages. Therefore, it is very difficult to identify log keys. Generally, the log messages printed by the same statement are often highly similar to each other, while two log messages printed by different log-print statements are often quite different. Based on this observation, we can use clustering techniques to group log messages printed by the same statement together, and then find their common part as the log key.

However, the parameters may cause some clustering mistakes because the log messages printed by different statements may also be similar enough if they contain a lot of identical parameter values. In order to reduce the parameters' influence on clustering, we first erase the contents that are obvious parameter values according to some empirical knowledge. Then, we further apply a raw log key clustering and group splitting algorithm to obtain log keys. Figure 1 gives an example to illustrate the procedure of extracting log keys from log messages.

A. Erasing parameters by empirical rules

As we know, parameters are often in forms of numbers, URIs, IP addresses; or they follow the special symbols such as the colon or equal-sign; or they are embraced by braces, square brackets, or Parentheses. These contents can be easily identified. Therefore, empirical rules are often used to recognize and remove these parameters [9]. By roughly going through the log files, we can define some empirical regular expression rules to describe those typical parameter cases, and erase the matched contents. After that, the left contents of log messages are defined as raw log keys. The second block of Figure 1 gives some examples of raw log keys. We can see that the IP addresses, the numbers, and the full path of a file are all removed from the log messages.

Although many parameters are erased, there are still some parameters that could not be completely removed in raw log keys. The main reason is that the empirical rules can't exhaust all parameter patterns without application specific knowledge.

B. Raw log key clustering

We separate a raw log key into words using a space as separator. We use words as primitives to represent raw log keys because words are minimal meaningful elements in a sentence. So, each raw log key can be represented as a word sequence.

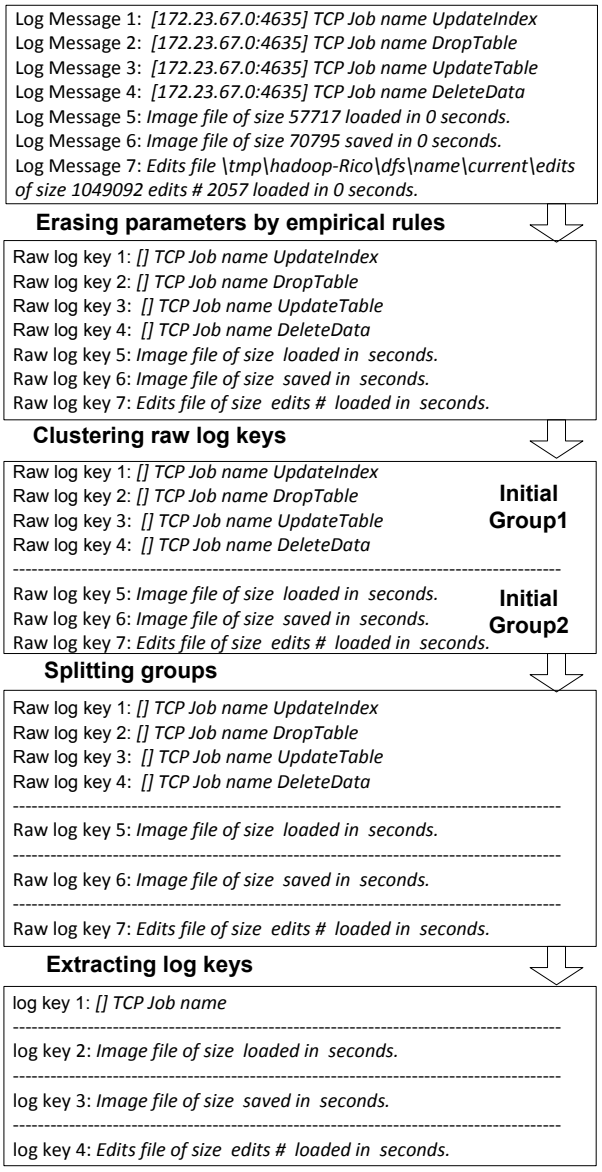
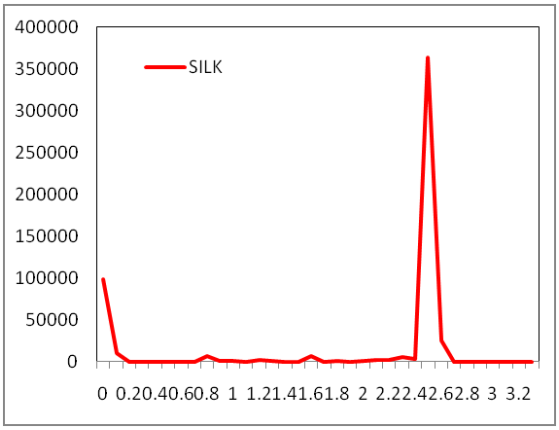


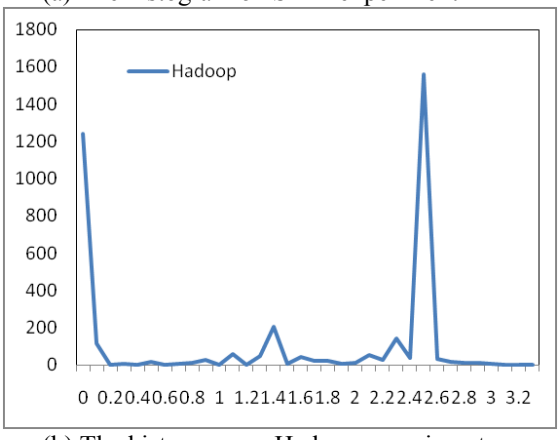
Figure 1 Examples of log key extraction

Before clustering, we need to find a proper metric to represent the similarity of two raw log keys. The string edit distance is a widely used metric to represent the similarity between word sequences. It equals to the number of edit operations required to transform one word sequence to the other. One edit operation can operate only one word. The operation can be adding, deleting or replacing. Obviously, the edit distance only counts the number of operated words; it does not consider the positions of the operated words. However, for our problem, the positions of operated words in raw log keys are meaningful for measuring similarity. It is because most programmers tend to write text messages (log keys) firstly, and then add parameters afterwards. Therefore, words at the beginning of raw log keys have

more probability to be parts of log keys than words at the end of raw log keys do. Therefore, the operated word at the beginning of the raw log keys should be more significant for measuring raw log keys' difference. Based on this observation, we measure raw log keys' similarity by the weighted edit distance, in which we use sigmoid similar function to compute weights at different positions. For two raw log keys rk_1 and rk_2, we denote the necessary operations required to transform rk_1 to rk_2 as $OA_1, OA_2, \ldots, OA_{EO}$; EO is the number of necessary operations. The weighted edit distance between rk_1 and rk_2 is denoted as $WED(rk_1, rk_2)$, $WED(rk_1, rk_2) = \sum_{i=1}^{EO} \frac{1}{1+e^{(x_i - \upsilon)}}$. Here, x_i is the index of the word that is operated by the i^{th} operation OA_i; υ is a parameter controlling weight function.



(a) The histogram on SILK experiment



(b) The histogram on Hadoop experiment

Figure 2. The histogram of raw log key pair number over weighted edit distance

We cluster similar raw log keys together. For every two log keys, if the weighted edit distance between them is smaller than a threshold ς, we connect them with a link. Then, each connected component corresponds to a group which is called as an initial group. The initial group examples are shown in the third block in Figure 1.

The threshold ς could be automatically determined according to the following procedure. For every two raw log keys, we compute the weighted edit distance between them. Then we obtain a set of distance values. Each distance should be either inner-class distance or inter-class distance. The inner-class (or inter-class) distance is the distance between two raw log keys corresponding to the same log key (or different log keys). In general, the inner-class distances are usually small while the inter-class distances are large. Therefore, we use a k-means clustering algorithm to cluster all distances into two groups. The distances in the two groups roughly correspond to the inner-class and the inter-class distances respectively. Finally, we select the largest distance from the inner-class distance group as the value of threshold ς.

We obtain the raw log keys by the experiments on Hadoop and SILK respectively (the experiments' details are described in section 7). We calculate the distances between every two raw log keys, and show the histogram of raw log key pair number over distance in Figure 2. The x-coordinate is the value of the weighted edit distance. The y-coordinate is the number of raw log key pairs. The figures show that: (1) There are two significant peaks in each histogram. It seems that the proposed weighted edit distance is a good similarity metric for raw log key clustering. (2) There is a flat region between two peaks. It implies that our raw log key clustering algorithm is not sensitive to the threshold ς.

C. *Group splitting*

Ideally, raw log keys in the same initial group correspond to the same log key. In such cases, a log key can be obtained by extracting the common part of the raw log keys in the same initial group. However, raw log keys in one initial group may correspond to different log keys because those log keys are similar enough. To handle those cases, we propose a group splitting algorithm to obtain log keys.

For an initial group, suppose there are GN raw log keys in this group. The common word sequence of the raw log keys within the group could be represented by $CW_1, CW_1, \ldots, CW_N$. For example, the initial group 2 in Figure 1 contains raw log key 5, 6, 7, and the common word sequence in the raw log keys are "***file***", "***of***", "***size***", "***in***", "***seconds***".

For each of the raw log keys in this group, e.g. the i^{th} log key, the common word sequence $CW_1, CW_2, \ldots, CW_N$ separates the raw log key into N+1 parts which is denoted as $DW_1^i, DW_2^i, \ldots, DW_N^i, DW_{N+1}^i$, where DW_j^i $(2 \le j \le N-1)$ is the i^{th} raw log key's content between CW_{j-1} and CW_j; DW_1^i is the i^{th} raw log key's content on the left side of CW_1; DW_{N+1}^i is the i^{th} raw log key's content on the right side of CW_N. We call DW_j^i as the private content at position j of the i^{th} raw log key. In the above example, the private content sequence of raw log key 7 is "***Edits***", $\emptyset$, $\emptyset$, "***edits*** #

loaded", ∅,∅. In the paper, ∅ represents that there is not any word in the private content.

For each position j, $1 \le j \le N+1$, we can obtain GN private contents at position j from GN raw log keys in the group, and they are DW_j^1, DW_j^2, …, DW_j^{GN}. We denote the number of different values (not including ∅) among those GN values as VN_j, and VN_j is called the private number at position j. For the initial group 2 in Figure 1, VN_1 =2, VN_2 =0, VN_3 =0, VN_4 =3, VN_5 =0, VN_6=0.

Intuitively speaking, if the private contents at position j are parameters, VN_j is often a large number because parameters may probably have many different values. However, if the private contents at position j are a part of log keys, VN_j should be a small number. Based on this observation, we find the smallest positive one among VN_1, VN_2,…, VN_N, VN_{N+1}, e.g. VN_J. If VN_J is equal to or bigger than a threshold ϱ, which means that the private contents at position J have at least ϱ different values, then we consider that the private contents at position J are parameters. In such a situation, this initial group does not split anymore. Otherwise, if VN_J is smaller than the threshold ϱ, we consider that the private contents at position J are a part of log keys. In such a situation, this initial group splits into VN_J sub-groups, satisfying that the raw log keys in the same sub-group have the same private content at position J. In the paper, we set ϱ as 4 according to experiments.

For the initial group 2, VN_1 is the smallest positive value 2 and is smaller than the threshold 4, so the initial group 2 splits into 2 sub-groups according to raw log keys' private contents at position 1. The raw log key 5 and 6 are in one sub-group, because they have the same private content "***Image***"; the raw log key 7 is in the other sub-group.

When there are multiple private numbers at different positions that have the same smallest positive value smaller than the threshold, we further compare the entropies at those positions respectively, select the one position with the minimal entropy, and split the group according to the private contents at that position. We denote the entropy at position j as EP_j. We compute EP_j according to the distribution of private content values at position j. For example, for the initial group 2 and j=1, we can obtain 3 values of the private content which are "***Image***", "***Image***", and "***Edits***". The value's distribution is p("***Image***")=2/3, p("***Edits***")=1/3, so $EP_1 = -\frac{2}{3}\log\frac{2}{3} - \frac{1}{3}\log\frac{1}{3} = 0.918$. The entropy rule is reasonable because a smaller entropy indicates lesser diversity, which means the private contents at that position have more possibility to be parts of log keys.

If there are still multiple positions that have the same private number and the same entropy, then we split the group according to the private contents at the most left one among those positions.

We perform the split procedure repeatedly, until there is no group satisfying the split condition. Finally, we extract the common part of raw log keys in each group as a log key.

D. Determine log keys for new log messages

After the above steps, we obtain the log key set from the training log messages in the training log files. When a new log message comes, we determine its log key according to the following two steps: First, we use the empirical rules to extract the raw log key from the log message. Second, we select the log key which has the minimal edit distance to the raw log key of the log message. If the weighted edit distance between the raw log key and the selected log key is smaller than a threshold σ, the selected log key is considered as the log key of the log message. Otherwise, the log message is considered as an error log message, and its log key is its raw log key. Here, we set σ as the largest one of the weighted edit distances between all raw log keys of training log messages and their corresponding log keys.

By replacing each log message with its corresponding log key, a log message sequence can be converted into a log key sequence.

IV. WORK FLOW MODEL

In order to detect anomalies of work flows, we use a Finite State Automaton (FSA) to model the execution behavior of each system module. Although there are some other alternate models, such as Petri-Net, we adopt FSA because it is simple but effective. FSA has been widely used in testing and debugging software applications [11]. A FSA consists of a finite number of states and transitions between the states. A set of algorithms have been proposed in previous literature to learn FSA from sequential log sequences [10, 11, 12]. In this paper, we use the algorithm proposed by [11] to learn a FSA for each system component from training log key sequences which are produced by normally completed jobs. Each transition in the learned FSAs corresponds to a log key. All training log key sequences can be interpreted by the learned FSAs. Therefore, each training log key sequence can be mapped to a state sequence. Figure 3 shows the example of the learned FSM of JobTracker of Hadoop (refer to Section 7.1). We give the state interpretations according to the log message in Table 1. From the learned the FSM, we obtain the following work flow: from S87 to S96, the JobTracker carries out some initialization tasks when a new job is submitted. After initialization, the state machine enters S197 to add a new Map/Reduce task. For each map task, it selects local or remote data source for processing. Then, the task is completed. When the last task is finished, the job is completed, and all resources of tasks are cleared iteratively. In fact, the learned FSM correctly reflects the real work flow of the JobTracker.

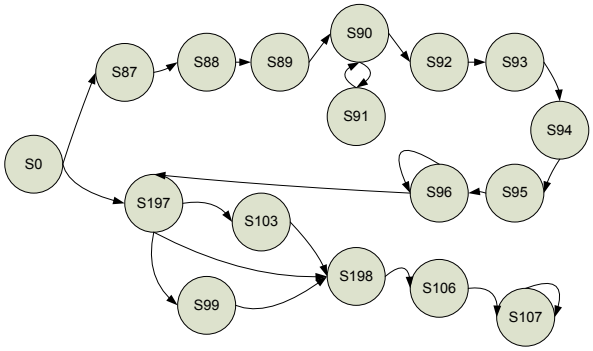


Figure 3. Example of a learned FSM

Table 1. The interpretations of states

State	Interpretation
S87~ S96	Initialization when a new job submitted
S197	Add a new map/reduce task
S103	Select remote data source
S99	Select local data source
S198	Task complete
S106	Job complete
S107	Clear task resource

V. PERFORMANCE MEASUREMENT MODEL

In this section, we present our technique to characterize the performance of the normally completed jobs. By comparing with normal performance characteristics, we can detect low performance in new jobs.

After log key extraction, we obtain corresponding log key sequences. The time stamp of a log key is the same as the time stamp of its corresponding log message. In order to derive a performance measurement model, we need to know applications' execution states. Therefore, we first convert each log key sequence to its corresponding state sequence. A state's time stamp is specified by the time stamp of its corresponding log key in the log key sequence.

In a system execution, there are two types of low performance problems. One is that the time interval that a system component transits from a state to the next state is much longer than normal cases; we name it transition time low performance. The other is that the circulation numbers of a loop structure are far more than normal cases; we name that loop low performance. We use the transition time between adjacent states and the circulation numbers of all loop structures to characterize the normal performance of jobs.

A. Transition time measurement model

In a distributed system, each machine writes log message sequences to its local disc independently. Therefore, different training state sequences may be derived from logs in different machines. Suppose we have *M* machines in a distributed system. For each state transition in the FSA, e.g. from S_a to S_b, the time intervals between two adjacent states (S_a, S_b) in the training state sequences produced by i^{th} machine are denoted as $\tau_i^1(S_a,S_b)$, $\tau_i^2(S_a,S_b)$,…, $\tau_i^{K_i}(S_a,S_b)$; $1 \le i \le M$. Here, K_i is the total number of the time intervals in all state sequences produced by the i^{th} machine.

We use a Gaussian model to present the distribution of the state transition interval. In practice, the computational capacity of machines in a distributed system is often heterogeneous. The different computing capacity of machines results in the state transition time intervals in different machines being quite different. In order to handle this problem, we introduce a capacity parameter for each machine. Our model contains machine independent Gaussian distribution parameters $\{\mu(S_a,S_b),\ \sigma^2(S_a,S_b)\}$ and machine dependent capacity parameters $\{\lambda_1(S_a,S_b),\lambda_2(S_a,S_b),\ldots,\ \lambda_M(S_a,S_b)\}$. Here, the Gaussian distribution $N(\mu(S_a,S_b),\sigma^2(S_a,S_b))$ is used to represent the distribution of the state transition time on an imaginary computer with a standard computing capacity. It is only determined by the property of the specified state transition, and does not depend on the property of any specific machine. The computers' properties are modeled by the computers' computing capacity parameters $\lambda_i(S_a,S_b)$, $1 \le i \le M$. The computers' computing capacity parameters are also associated with the state transition, because different state transitions often correspond to different computing tasks and the same computer may have a different computing capacity under different work load characteristics.

In this subsection, because the state transition is specified, we abridge state indicators in expressions or formulas for simplicity. We assume that the mean value of state transition time in the i^{th} machine is proportional to its computing capacity parameter λ_i, and the variance is proportional to λ_i^2. With that assumption, the obtained transition time instances in the i^{th} machine satisfy the Gaussian distribution $N(\lambda_i\mu,(\lambda_i\sigma)^2)$, $1 \le i \le M$. We further assume that the obtained transition time instances are independent, and then the likelihood function is as follows.

$$p\left(\tau_1^1,\tau_1^2,\ldots,\tau_1^{K_1},\tau_2^1,\tau_2^2,\ldots,\tau_2^{K_2},\ldots,\tau_M^1,\ldots,\tau_M^{K_M}\right) = \prod_{i=1}^{M}\left[\prod_{j=1}^{K_i} N(\tau_i^j;\lambda_i\mu,(\lambda_i\sigma)^2)\right] \quad (1)$$

With the variable substitutions of $\alpha_i = \lambda_i\mu$ and $\beta = \frac{\sigma^2}{\mu^2}$, we can obtain the log-likelihood function:

$$L(\alpha_1,\alpha_2,\ldots,\alpha_M,\beta) = -\sum_{i=1}^{M}\sum_{j=1}^{K_i}[2\ln\alpha_i + \ln\beta + \frac{1}{\beta}(1-\frac{\tau_i^j}{\alpha_i})^2] \quad (2)$$

According to the Maximum Likelihood Estimation criterion, the optimal parameters should maximize $L(\alpha_1,\alpha_2,\ldots,\alpha_M,\beta)$. Because the optimal parameters should satisfy that the partial differentiates of $L(\alpha_1,\alpha_2,\ldots,\alpha_M,\beta)$ equal to 0, we have:

$$\begin{cases} \alpha_i = \dfrac{\sqrt{\left(\sum_{j=1}^{K_i}\tau_i^j\right)^2 + 4K_i\beta\left(\sum_{j=1}^{K_i}{\tau_i^j}^2\right)} - \left(\sum_{j=1}^{K_i}\tau_i^j\right)}{2K_i\beta}, 1 \le i \le M \\ \beta = \left(\sum_{i=1}^{M}\sum_{j=1}^{K_i}(\alpha_i - \tau_i^j)^2\right) / \left(\sum_{i=1}^{M} K_i {\alpha_i}^2\right) \end{cases} \quad (3)$$

However, there is no closed form solution to the above equation group; we can only use an iterative procedure to obtain an approximation of the optimal parameters. It could be proved that after each iteration step, the value of $L(\alpha_1, \alpha_2, \ldots, \alpha_M, \beta)$ increases. The iterative procedure is shown in Table 2. When the difference of β in two iterations is small enough ($< \mathrm{Th}_\beta$), the iterative procedure terminates.

Finally, we can obtain the transition time measurement model: the transition time from S_a to S_b in the i^{th} machine satisfies the Gaussian distribution $N(\alpha_i(S_a, S_b), \alpha_i^2(S_a, S_b)\beta(S_a, S_b))$.

It should be pointed out that the above algorithm can be easily implemented in a parallel mode. According to formula (3), when given β, α_i can be determined by the sample data in the i^{th} machine, i.e. τ_i^j $(1 \le j \le K_i)$. Thus, each α_i can be calculated separately at the i^{th} machine. When given α_i $(1 \le i \le M)$, the intermediate results, i.e. ${K_i}{\alpha_i}^2$ and $\sum_{j=1}^{K_i}(\alpha_i - \tau_i^j)^2$, can also be calculated by machines separately. Then, it is very easy to integrate those intermediate results to obtain β. Therefore, our algorithm can be used to learn models from the logs of very large scale systems.

B. Circulation numbers measurement model

The circulation numbers of loop structures are meaningful measurements for low performance detection because some executions' low performance is caused by abnormally more loops although each of its adjacent state transition times seem normal. A loop structure is defined as a directed cyclic chain composed by the state transition in the learned FSA. For example, for the FSA shown in Figure 4, one loop structure is $\{S_2, S_3\}$, the other is $\{S_1, S_2, S_3\}$. A loop structure execution instance is formed by consecutively repeating several rounds of a loop structure from its beginning to its end; and the number of execution rounds is defined as a circulation number. For example, in the state sequence "$S_0 S_1 S_2 S_3 S_2 S_3 S_1 S_2 S_3 S_4$", the subsequences, e.g. "$S_2 S_3 S_2 S_3$" and "$S_2 S_3$", are two execution instances of the loop structure $\{S_2, S_3\}$ in the state sequence, and the circulation numbers are 2 and 1 respectively; the subsequence, e.g. "$S_1 S_2 S_3 S_2 S_3 S_1 S_2 S_3$" is an execution instance of the loop structure $\{S_1, S_2, S_3\}$, and the circulation number is 2.

We identify loop structures in the learned FSA. For each loop structure, e.g. L, we find the execution instances of L in all training state sequences, and record their circulation numbers as $C^1(L), C^2(L), \ldots, C^{H(L)}(L)$; where $H(L)$ is the amount of L's execution instances. Similarly, we use Gaussian distribution $N(\mu(L), \sigma^2(L))$ to model them.

$$\mu(L) = \frac{1}{H(L)}\sum_{i=1}^{H(L)} C^i(L) \quad (4)$$

$$\sigma^2(L) = \frac{1}{H(L)}\sum_{i=1}^{H(L)} [C^i(L) - \mu(L)]^2 \quad (5)$$

Table 2. Iterative procedure to compute parameters

Initialization:

$\alpha_i = \frac{1}{K_i}\sum_{j=1}^{K_i}\tau_i^j$, $1 \le i \le M$;

$\beta = \beta' = 0$;

While *true*

Set $\beta' = \beta$;

Using current value of α_i $(1 \le i \le M)$, compute β according to the last one formula in the formula group (3);

If $|\beta' - \beta| < Th_\beta$,

break;

Else

using current value of β, compute α_i $(1 \le i \le M)$ according to the first M formula in the formula group (3);

Endif

End

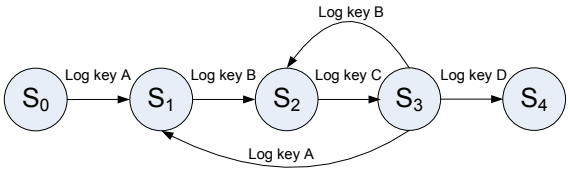


Figure 4. An example of FSA

VI. ANOMALIES DETECTION

For a newly input log message sequence, we can obtain the corresponding log key sequence according to section 3.4. If the log key sequence can be generated by the learned FSA, then we consider that there is no work flow error. Otherwise, the first log key in the sequence that can't be generated by the learned FSA is detected as a work flow error. The details of work flow error detection can be found in paper [11]. In this paper, we mainly focus on the low performance detection.

A. Transition time low performance detection

During low performance detection, we first convert the testing log key sequences to the corresponding state sequences according to the learned FSA. For each state transition in the state sequence produced by the i^{th} machine, e.g. from S_a to S_b, we then compare its execution time with the learned transition time measurement model of the i^{th} machine. If the execution time is larger

than a threshold $\gamma_i(S_a, S_b)$, it is considered as a transition time low performance. Here, the threshold is defined as the sum of the mean value and ϵ times standard deviation of the learned transition time distribution.

$$\gamma_i(S_a, S_b) = \alpha_i(S_a, S_b)(1 + \epsilon\sqrt{\beta(S_a, S_b)}) \quad (6)$$

Obviously, the smaller ϵ is, the more state transitions are detected as low performance problems. At the same time, there will be more false positives and less false negatives. When applying our technique, users can adjust the value of ϵ according to real requirements. In the experiments, we set ϵ as 3.

B. Loop low performance detection

Similar to transition low performance detection, for each loop structure L, we calculate its threshold $\vartheta(L)$ as follows

$$\vartheta(L) = \mu(L) + \epsilon\sigma(L) \quad (7)$$

We find the execution instances of L whose circulation numbers are larger than $\vartheta(L)$ as loop low performance.

VII. EXPERIMENTS

In this section, we evaluate the proposed technique through detecting anomalies in two typical distributed computing systems: Hadoop and SILK (a privately owned distributed computing system). In this section, we represent some typical cases to demonstrate our technique, and give out some over all evaluations on our experiment results.

A. Case study on Hadoop

Hadoop [13] is a well-known open-source implementation of Google's Map-Reduce [14] framework and distributed file system (GFS)[15]. It enables distributed computing of large scale, data-intensive and stage-based parallel applications. Hadoop is designed with master-slave architecture. NameNode is a master of the distributed file system, which manages the metadata of all stored data chunks, while DataNodes are slaves used to store the data chunks. JobTracker acts as a task scheduler that decomposes the job into smaller tasks and assigns the tasks to different TaskTrackers. A TaskTracker is a worker of a task instance.

The logs produced by Hadoop are not sequential log message sequences in its original forms. The log messages for different tasks interleave together. However, we can easily extract sequential log message sequences from logs by the task IDs.

Table 3. Basic configurations of machines

Machine	Basic configuration
PT03~PT05	Intel dual-core E3110@3.0G, 8G RAM
PT06~PT11	Intel quad-core E5430@2.66G, 8G RAM
PT12~PT17	AMD Quad-Core 2376@2.29G, 8G RAM

Our test bed of Hadoop (version 0.19) contains 16 machines (from PT3 to PT17) connected with a 1G Ethernet switch. The basic configurations are listed in Table 3. Among them, PT17 is used as a master that hosts NameNode and JobTracker components. The others are used as slaves, and each slave hosts DataNode and TaskTracker components. During the experiments, we run the stand-alone program (namely CPUEater) which consumes a predefined ratio of CPU so that we can better simulate a heterogeneous environment. Table 4 shows the utility ratios (i.e. 100%-consumed CPU ratio of CPUEater) and the learned model parameters. We can see that the more powerful machine, the smaller the average transition time is.

Table 4. Utility ratio and model parameters

Machine	Utility Ratio	Learned parameters	
		α (s)	β
pt09	100%	38.04	0.0187
pt07	30%	47.10	
pt12	50%	65.02	
pt14	30%	65.63	
pt05	50%	78.46	

In the learning stage, we run 100 jobs of counting words in the test bed and collect the produced log files of these jobs as training data. The counting words job gives out the word frequency in the input text files. Each input text file for a job is about 10G. In the testing stage, we run 30 counting words jobs to produce testing data.

In this subsection, we give one example of the test cases in Table 5. In this case, we manually insert a low performance problem by limiting the bandwidth of machine PT9 to 1Mbps when running a job, and check whether our algorithm can detect it. The result shows that our algorithm can successfully detect the low performance problem that the transition time from state #21 to state #1 is much larger than the normal cases (i.e. 60s > 38.04s).

Table 5. Low performance transition of Hadoop

Time Stamp	State ID	State Meaning
2009-01-18 10:42:31.452	21	Data source for a Map task is selected.
2009-01-18 10:43:30.423	1	Map task is completed.

B. Case study on SILK

SILK is a distributed system developed by our lab for large scale data intensive computing. Unlike MapReduce, SILK uses a Directed Acyclic Graph (DAG) framework similar to Dryad [16]. SILK is also designed based on the master-slave architecture. A Scheduler-Server component works as a master to decompose the job into smaller tasks, and then schedule and manage the tasks. SILK produces many log files during execu-

tion. For example, it generates about 1 million log messages every minute (depending on workload intensity) in a 256-machine system. Each log message contains a process ID and a thread ID. We can group log messages with the same process ID and thread ID into sequential log sequences. The test bed of SILK contains 7 machines (1 master, 6 slaves), which is set up for daily-build testing. As our training data, we collect the training log files of all successful jobs during a ten-day running in the test-bed. The test logs are generated during one month of daily-build testing. Our algorithm can detect several system execution anomalies (shown in Table 7). In this subsection, we give two typical examples.

Case 1: In this case, due to a networking issue, a slave task (CopyDatabase) tries several times to connect to a database, which makes a response to the master (SchedulerServer) and is largely delayed. From the log sequence of the master, our algorithm finds that the transition time from state #424 to state #428 is much larger than expected (see Table 6). According to the learned model, the average time interval is 12.32s, while the time interval in this case is 42.53s. Therefore, our algorithm detects it as an anomaly of transition time low performance.

Table 6. Case 1: Low performance transition of SILK

Time Stamp	State ID	State Meaning
2008-09-09 18:44:52.749	424	Job task is started.
2008-09-09 18:45:35.280	428	A worker progress event is received.

Case 2: In this case, the master (SchedulerServer) sends a job finish message to a client, but the client never replies. This causes the master to repeat the attempt more than 20 times before giving up. Compared with 1 in normal situations, it is detected as a loop low performance anomaly by our algorithm.

C. Overall results

Table 7 shows the overall results of anomaly detection on Hadoop and SILK. In the experiments on Hadoop, we detect 15 types of anomalies, 2 of them being false positives (FP). In the experiments on SILK, we detect 91 types of anomalies, 22 of which are FPs. Looking into these FPs, we find that our current loop low performance detection is sensitive to different workloads. This is because the circulation numbers of some loop structures largely depend on the work load. With the help of user's feedback, such FPs can be reduced by relaxing the threshold ϵ.for the corresponding loop structures.

D. Comparison of log key extraction

In order to evaluate our log key extraction method, we compare our method with the method proposed by Jiang et. al. [9]. The comparison results are shown in Table 8, where the numbers of real log key types are manually identified, and are used as the ground truth. For our algorithm, the numbers of obtained log key types are very close to the ground truth. Furthermore, more than 95% of the log keys extracted by our method are identical with the real log keys. By comparison, our algorithm significantly outperforms the algorithm of [9].

Table 7. Overall evaluation results

Anomaly type	Hadoop		SILK	
	Detected anomaly types	False positive	Detected anomaly types	False positive
Work flow error	4	0	16	0
Transition time low performance	6	0	6	0
Loop low performance	5	2	69	22

Table 8. Comparison results of log key extraction

System	Extracted log key types of Jiang et.al [9]	Extracted log key types of our method	Real log key types
Hadoop	257	197	201
SILK	2287	651	631

VIII. CONCLUSION

As the scale and complexity of distributed systems continuously increases, the traditional problem of diagnosis approaches; experienced developers manually checking system logs and exploring problems according to their knowledge becomes inefficient. Therefore, a lot of automatic log analysis techniques have been proposed. However, the task is still very challenging because log messages are usually unstructured free-form text strings and application behaviors are often very complicated.

In this paper, we focus on the log analysis technique for automated problem diagnosis. Our contributions include: (1) We propose a technique to detect anomalies, including work flow errors and low performance, by analyzing unstructured system logs. The technique requires neither additional system instrumentation nor any application specific knowledge. (2) We propose a novel technique to extract log keys from free text messages. Those log keys are the primitives in our model used to represent system behaviors. The limited number of log key types avoids the curse of dimension in the statistic learning procedure. (3) Model the two types of low performance. One is for modeling execution time of state transitions; the other is for modeling the circulation number of loops. In the model, we take into account the factors of heterogeneous environments. (4) The detection algorithm can remove false positive detection of low performance caused by inputting large

workloads. Experimental results on Hadoop and SILK demonstrate the power of our proposed technique.

Future research directions include utilizing log parameter information to conduct further analysis, performing analysis on parallel logs that are produced by multi-thread or event based systems, visualizing the models and the anomalies detection results to give intuitive explanation for human operators, and designing a user-friendly interface.

IX. REFERENCES

[1] W. Dickinson, D. Leon, and A. Podgurski, "Finding Failures by Cluster Analysis of Execution Profiles. In the proceeding of the 23rd International Conference on Software Engineering, May, 2001.

[2] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller, "Problem Diagnosis in Large-Scale Computing Environments", In the Proceedings of the ACM/IEEE SC 2006 Conference, Nov. 2006.

[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Mining Console Logs for Large-Scale System Problem Detection", In Workshop on Tackling Computer Problems with Machine Learning Techniques, Dec. 2008.

[4] C. Yuan, N. Lao, J.R. Wen, J. Li, Z. Zhang, Y.M. Wang, and W. Y. Ma, "Automated Known Problem Diagnosis with Event Traces", In the proceeding of EuroSys 2006, Apr. 2006.

[5] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore, "Investigation of Failure causes in work-load driven reliability testing", In the proceeding of the 4th International Workshop on Software Quality Assurance, Sep. 2007.

[6] S. Orlando and S. Russo, "Java Virtual Machine Monitoring for Dependability Benchmarking", In proceedings of the 9th IEEE International Symposium on Object and Component-oriented Real –time Distributed Computing, Apr. 2006.

[7] 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.

[8] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. "Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata", in the proceeding of 2nd International Conference on Autonomic Computing, Jun. 2005.

[9] Z. M. Jiang, A. E. Hassa, P. Flora, and G. Hamann, "Abstracting Execution Logs to Execution Events for Enterprise Applications", in the proceeding of the 8th International Conference on Quality Software (QSIC), pp.181-186, 2008.

[10] G. Ammons, R. Bodik, and J. R. Larus, "Mining Specifications", in the proceeding of ACM Symposium on Principles of Programming Languages (POPL), Portland, Jan. 2002.

[11] L. Mariani and M. Pezz`e, "Dynamic Detection of COTS Components Incompatibility", IEEE Software, pp. 76-85, vol.5, 2007.

[12] D. Lo, and S.-C. Khoo, "QUARK: Empirical Assessment of Automaton-based Specification Miners", in proceeding of the 13th Working Conference on Reverse Engineering (WCRE'06), 2006.

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

[14] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", In the proceeding of USENIX Symposium on Operating Systems Design and Implementation (OSDI), Dec. 2004.

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

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks", In the proceeding of EuroSys, Mar. 2007.

[17] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff, "Mining for Misconfigured Machines in Grid Systems", in Proceeding of 12th ACM International Conference on Knowledge Discovery and Data Mining (KDD), pp. 687-692, Philadelphia, PA, USA, 2006.

[18] M. Chen, A.X. Zheng, J. Lloyd, M. I. Jordan, E. Brewer, "Failure Diagnosis Using Decision Trees", in the processing of the first International Conference on Autonomic Computing (ICAC), pp. 36-43, 2004.