Mining Dependency in Distributed Systems through Unstructured Logs Analysis

Jian-Guang LOU[1], Qiang FU[1], Yi WANG[2], Jiang LI[1]
[1]Microsoft Research Asia; [2]Beijing University of Posts and Telecommunications

Abstract

Dependencies among system components are crucial to locating root errors in a distributed system. In this paper, we propose an approach to mine inter-component dependencies from unstructured logs. The technique requires neither additional system instrumentation nor any application specific knowledge. In the approach, we first parse each log message into its log key and parameters. Then, we find dependent log key pairs belong to different components by leveraging co-occurrence analysis and parameter correspondence. After that, we use Bayesian decision theory to estimate the dependency direction of each dependent log key pair. We further apply time delay consistency to remove false positive detections. Case studies on Hadoop show that the technique successfully identifies the dependencies among the distributed system components.

Keywords: Log analysis, dependency graph, co-occurrence analysis, root error localization

1 Introduction

In a distributed system, dependencies are numerous, complex, and inherent, spanning system components across the network and the infrastructure. Errors often propagate across distributed components due to inter-component dependencies. Understanding the dependencies among components is critical to locating the root errors from a set of related errors. The understanding not only lets administrators quickly locate the root errors, but also provides them cues to learn the problems. Several research efforts [1, 2, 3] have been made to discover dependencies for system management. However, most of research efforts focus on mining the service level dependency based on communication traces.

In this paper, we propose an approach to discover dependencies among a set of distributed components based on console logs printed by a system during its execution. Console logs are often produced and collected for troubleshooting, work load analysis, and system behavior tracking. They often contain a wide variety of information about system behavior including system events, state changes and inter-component interactions. However, log messages are often in the form of unstructured text strings, and the clock of the machines that produce logs may not be synchronized precisely. Therefore, it is very challenging to mine dependencies through unstructured logs.

Our approach requires neither additional system instrumentation nor any application specific knowledge, and mines dependencies in a black-box manner. In the approach, we first parse each log message into the log key and parameters (refer to Section 3), and then, mine the dependencies among different components by leveraging co-occurrence analysis, parameter correspondence and time information. Finally, we show how the dependencies can help us to locate problems in a distributed system, and to reveal the error propagation diagram.

In our technique, inter-component dependencies are learned based on the cues gained from the previous jobs' logs (namely training logs). We assume that each log item has a corresponding time stamp. We further assume that the system clocks of different hosts are roughly synchronized. This is reasonable because Network Time Protocol (NTP) has been a built-in service in most operating systems including Windows and Linux.

The paper is organized as follows. In Section 2, we briefly survey several related research efforts. In Section 3, we briefly present how to parse log messages into log keys and parameters. Then, the algorithm that mines dependencies from logs is described in Section 4. In Section 5, some primitive results are presented. We give a brief introduction on how to use the obtained dependencies to locate the root errors in Section 6. Finally, in Section 7, we conclude the paper.

2 Related Work

Understanding the dependencies among different components of a large-scale distributed system is extremely important for problem diagnosis. A lot of previous research efforts have been put on mining dependencies in a black-box manner.

Most of the research efforts focus on service level dependencies among different network services from communication traces [1, 2, 4, 5, 6]. In [6], the authors introduce the Sherlock system to discover an Inference Graph to model the dependencies among different

network services, and then use the learned results to automatically localize problems. Sherlock simply uses co-occurrence probabilities to estimate dependencies. In [1] and [4], the authors proposed a distributed approach (Constellation) and a centralized method (AND) to construct a Leslie Graph, which estimates service level dependencies by inferring traffic correlation. Kandula et al [5] propose a system called eXpose, which uses the JMeasure (a known metric in the data-mining community) as a score to find the dependent service pairs. All above algorithms are all sensitive to the time window threshold. In [2], Chen et al introduce a dependency discovery technique based on traffic delay distributions, and find that their algorithm is much better than the previous ones. In [7], Kannan et al utilize a Poisson process to model the normal connections' arrival. Two connections with very small inter-arrival time interval are recognized as two dependent connections.

Besides the approaches that mine service level dependencies from traffic traces, Gupta et al propose a method to identify dependencies among components based on synchronized application events collected by a monitoring server [3, 8]. The method is realized by finding out the pattern that an activity period of component A contains some activity period of component B.

All above algorithms consider either the communication traces [1, 2, 4, 5, 6, 7], or the synchronized event traces [3, 8]. Differing from these previous algorithms, our algorithm tries to mine the inter-component dependencies based on unstructured logs of a distributed system. These unstructured logs are collected from different machines, and the time stamps of the logs may not be precisely synchronized. Fortunately, log messages often contain parameter information to help problem diagnosis (e.g. request tracing), therefore we can utilize the parameter information for our dependency mining.

3 Log Message Parsing

A log message usually records an event, state change or inter-component interaction of a run-time system component. It often contains two types of information: one is a free-form text string that is used to describe the semantic meaning of a recorded program event; the other is parameters that are used to express some important characteristics of the current task/request.

Because the two types of information have quite different meanings and functions, we define two different concepts, i.e. log key and parameter, to represent the two types of information in log messages respectively. A log key is defined as the common content of all log messages that are printed by the same log-print statement in the source code. Parameters are defined as the printed values of variables in the log-print statement. For example, for each log message printed by the following log print statement (in C language), its log key is "***the Job id is starting!***", and the parameter is the value of the variable "***JobID***".

fprintf(Logfile, "the Job id %d is starting!\n", JobID);

However, because the source code is often not available, we do not know what are log keys and parameters in log messages. In this paper, we use the algorithm presented in paper [16] to separate log keys and parameter values from log messages. As we know, some parameters are in forms of numbers, URIs, IP addresses; or they follow the special symbols such as "=". These contents can be easily identified. We first extract the contents from log messages which are obvious parameter values according to some empirical knowledge. The remained parts of log messages are categorized to a set of raw log keys. Because the empirical rules can't extract all parameters completely so that raw log keys may still contain parameters, we further apply a clustering algorithm on these raw log keys to obtain a set of clusters. The common string in each cluster is considered as a log key, and other parts are recognized as parameters. The algorithm does not need any application specific knowledge. It can achieve an accuracy of more than 95%.

For a log message m, we denote the extracted log key as $K(m)$, the number of parameters as $PN(m)$, the i^{th} parameter's value as $PV(m,i)$. After log key and parameter extraction, each log message m with the time stamp $T(m)$ can be represented by a multi-tuple $[T(m), K(m), PV(m,1), PV(m,2), \ldots, PV(m,PN(m))]$, we call such multi-tuples as the tuple-form representations of the log messages.

4 Component dependency mining

After log key and parameter extraction, we then mine dependencies from the tuple-form representations of the log messages. In our paper, the dependencies mean the causal relationships between the log messages of different components. For example, in Hadoop, a TaskTracker prints a log message of "Task attempt_xx is done" when a map task is finished, and then, JobTracker prints a log message of "Task attempt_xx has completed task_xx successfully." It is obvious that the occurrence of the first log message causes the occurrence the second log message. The log messages' causal relations are the results of execution

logic expressed in the source code.

The execution instances of the same component running at different machines often produce the same set of logs, and have the same set of dependencies. For example, in Hadoop, TaskTrackers running at different slave machines are instances of the same TaskTracker component, and often produce the same set of logs. In this paper, we aggregate the logs produced by the same component's running instances that are distributed in multiple machines for dependency mining. In other words, the learned dependencies are the dependencies between log keys of different components. Our dependency mining algorithm is based on the following two observations:

- Co-occurrence observation: If event B depends on event A, then B is likely to occur within a short interval (namely *dependency interval,* denoted as τ_d) after A's occurrence.
- Correspondence observation: For most systems, two dependent logs often contain at least one identical parameter, such as request ID, which can help operator to track the execution flow. The correspondence of parameters can help us to find dependencies and largely reduce false positives.

For a dependent log pair "A causes B" that are produced by different machines, the temporal order of "A prior B" may not be correctly observed due to the time difference of machines. In order to overcome the possible temporal disorder of log message pairs, we derive inter-component dependencies through two steps. First, we evaluate whether two log keys have correlated occurrences and parameter correspondences in the logs (namely related pair). If two log keys are identified as a related pair, then, we estimate their dependency direction based on Bayesian decision theory.

4.1 Identification of Related Log Key Pair

We evaluate the co-occurrence of two log keys s and q and the correspondence of their parameters $PV(s,d_1)$ and $PV(q,d_2)$ based on the conditional probabilities $P(Q|q)$ and $P(Q|s)$. Here, Q represents the quadruple (s, d_1, q, d_2), $P(Q|q)$ is the probability that log key s occurs within a dependency interval around the occurrence of q, and the d_1^{th} parameter of s is equal to the d_2^{th} parameter of q , and it can be estimated through the following equation:

$$P(Q|s) = \frac{C_s(Q)}{O(s)}$$

where $O(s)$ is the number of all log messages whose log key is s, and $C_s(Q)$ is the total number of log messages (denoted as A) in all log files that satisfy the following two rules:

- $K(A) = s$;
- There exists at least a log message B satisfying that $K(B)=q$, $|T(A)-T(B)| < \tau_d$, and $PV(A,d_1)=PV(B,d_2)$. Here, τ_d is the *dependency interval.* For each A, all such log messages B form a set, denoted as $\Omega(A,Q)$.

Similarly, $P(Q|q)$ can also be estimated through the same procedure. Based on the conditional co-occurrence probabilities, we identify each related log key pair by assuming that at least one conditional probability of quadruple is higher than a threshold Th_{cp} :

$$\max_{d_1,d_2}\big(P(s,d_1,q,d_2|s), P(s,d_1,q,d_2|q)\big) \geq Th_{cp}$$

However, it is time consuming to calculate the conditional probabilities of all quadruples because there are too many quadruples. For example, if there are N log keys, and each log message has M parameters, we will have about $N(N-1)M^2$ quadruples. In order to improve the computational efficiency of the algorithm, we take the following steps.

First, we only estimate the above conditional probabilities for inter-component log key pairs, because the inter-component dependencies are more interested in the system management and fault localization.

Second, we carry out a pre-process to filter out some log key pairs that are obviously independent. We estimate the conditional concurrency probability of $P(q|s)$ which is the probability that log key s occurs in a dependency interval around the occurrence of log key q:

$$P(q|s) = \frac{C(s,q)}{O(s)}$$

where $C(s,q)$ denotes the number of log messages l in all log files that satisfy the following two rules (where τ_d is the dependency interval):

- $K(l) = s$;
- There exists at least one log message l' satisfying $|T(l) - T(l')| < \tau_d$ and $K(l') = q$.

Similarly, the conditional probability $P(q|s)$ is estimated through the same procedure. Then, if both $P(s|q) < Th_{cp}$ and $P(q|s) < Th_{cp}$ are true, we do not need to calculate the conditional probabilities of all quadruples of this log key pair, because $C_s(Q)$ is always not larger than $C(s,q)$.

4.2 Determine the dependency direction

For a dependent log key pair, in general, if s depends on q, then the log message of s should occur later than the log message of q. However, because log messages are usually printed at different machines,

the time stamps of log messages are recorded as the local time of the machines, which are often not precisely aligned. Therefore, the time stamp of the two dependent log messages may be disordered that makes it difficult to determine the real dependency directions of the related log key pairs. In the paper, we determine the dependency directions based on the Bayesian Decision theory.

Given a related log key pair (s, q), we can find n log message samples of the pair from the training log files (s_i, q_i), $i = 1 \cdots n$, and their corresponding time stamp pairs (t_{s_i}, t_{q_i}), $i = 1 \cdots n$. Because the log time stamps t_{s_i} and t_{q_i} are recorded as local time, we have the following equation:

$$t_{s_i} = \hat{t}_{s_i} + \delta_{s_i} \; and \; t_{q_i} = \hat{t}_{q_i} + \delta_{q_i}$$

where $\hat{t}_{s_i}$ and $\hat{t}_{q_i}$ are the absolute occurrence time of s_i and q_i respectively, δ_{s_i} and δ_{q_i} are the time alignment errors respectively. Therefore, delays of the dependent log keys (s, q) satisfy:

$$\frac{\sum_{i=1}^{n}(t_{s_i} - t_{q_i})}{n} = \frac{\sum_{i=1}^{n}(\hat{t}_{s_i} - \hat{t}_{q_i})}{n} + \frac{\sum_{i=1}^{n}\delta_{s_i} - \sum_{i=1}^{n}\delta_{q_i}}{n}$$

Let δ_{s_i} and δ_{q_i} , $i = 1 \cdots n$ be independent and identically distributed (i.i.d.) random errors with a mean of $E(\delta) = \mu$ and a variance of $var(\delta) = \sigma^2$. For example, in Map-Reduce [14], Map and Reduce components are randomly distributed to all computing nodes. According to the Central Limit Theorem, the following statistic

$$\mathrm{Y} = \frac{\sum_{\mathrm{i}=1}^{\mathrm{n}}\delta_{s_i} - \sum_{\mathrm{i}=1}^{\mathrm{n}}\delta_{q_i}}{\mathrm{n}}$$

asymptotically conforms to a normal distribution with a mean of zero and a variance of $\frac{2\sigma^2}{\mathrm{n}}$. Denoting $\frac{\sum_{i=1}^{n}(t_{s_i} - t_{q_i})}{n} = \mu_{sq}$ and $\frac{\sum_{i=1}^{n}(\hat{t}_{s_i} - \hat{t}_{q_i})}{n} = \hat{T}_{sq}$, we can find that $\hat{T}_{sq}$ asymptotically complies with a normal distribution with a mean of μ_{sq} and a variance of $\frac{2\sigma^2}{\mathrm{n}}$, if we have enough training log sequences. Based on the Bayesian Decision theory, we can determine the dependency direction as follows:

$$\mu_{sq} > \beta \;\rightarrow\; \hat{T}_{sq} > 0 \rightarrow s \; \boldsymbol{depends\ on} \; q$$

or

$$\mu_{sq} < -\beta \;\rightarrow\; \hat{T}_{sq} < 0 \rightarrow q \; \boldsymbol{depends\ on} \; s$$

The threshold β is used to control the confidence of the decision. In this paper, we simply set $\beta = 0.005$ seconds.

4.3 *Variance based False Positive Reduction*

In order to further refine the detected dependencies, we remove false positive dependencies by utilizing the delay information. Our idea is based on the observation that the delay's variance of a false positive is often much larger than that of a true positive. The reason is that the two log keys of a false positive co-occur by chance, and their delay values are often quite random. On the contrary, the delay distribution between dependent events often exhibits a typical spike [2]. In our implementation, we remove the detected dependencies if the variance of delay is larger than a threshold.

4.4 *Dependency Pruning*

During the above stage, we identify the dependent pairs through exploiting the concurrency of the log keys. Some redundant dependent pairs are found. For example, in Fig.1, because the log messages s_0 and s_1 sometime are printed in a very short time period, we can find two dependencies, D_1 and D_2, simultaneously. Similarly, D_3 and D_4 may also be found by our algorithm. In fact, D_2 and D_3 are redundant dependencies in these two cases because they can be inferred from D_1 and D_4 respectively. In order to obtain a compact and clear dependency graph, we carry out a pruning operation. It is set as the last step of our dependency learning process to cut the redundant dependency edges.

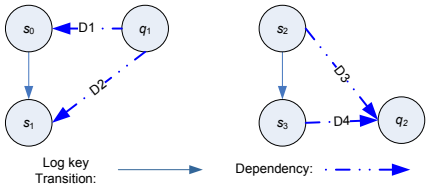

Figure 1: Dependency pruning.

5 Experimental Results and Discussion

In this section, we show the results of our algorithm through mining the dependencies from the Hadoop logs. Hadoop [13] is a well-known open-source implementation of Google's Map-Reduce [14] framework and distributed file system [15]. Hadoop is public available, so the results in this paper are verifiable and reproducible.

Our Hadoop test bed (v0.19) consists of 16 machines. One machine is used as a master to run NameNode and JobTracker. The rest machines are slaves to host DataNode and TaskTracker. We apply our algorithm to analyze the logs collected by 10 jobs. Each job counts the word number in a large text file with a size of 10G bytes. We first use loose parameters, the time window size τ as 3 seconds and the probability threshold Th_{cp} as 0.5, to obtain a set of detected dependencies (about 150 dependencies are found). Then, we manually verify whether they are true or not. We use

these labeled dependent log key pairs as our ground truth.

Fig.2 shows the rate of false positive and false negative dependencies as the value of τ increases from 0.1 seconds to 3 seconds. From this figure, we can find that when the time window size is larger than 1 second, the false negative rate does not decrease further. At the same time, the curve of false positive also becomes flat. Probability thresholds of 0.8 and 0.9 have similar false positive and false negative rate for most time windows values. Therefore, τ around 1 second is a good parameter. Note: the refinement of section 4.3 is not used in the experiment of Fig.2.

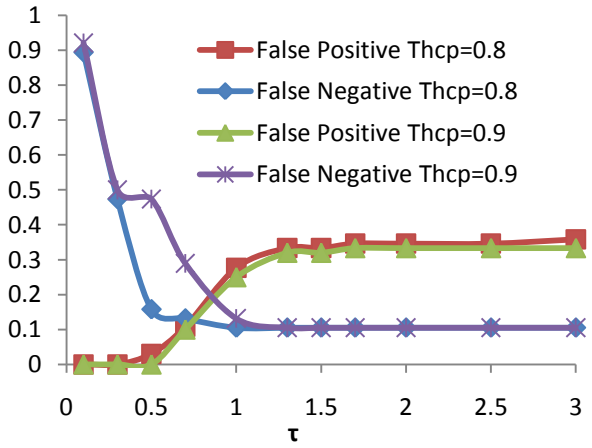

Figure 2. False positive and false negative rate as the time window size increases.

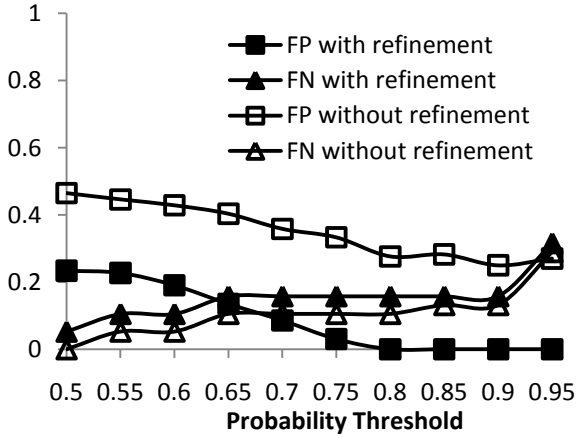

Figure 3. False positive (FP) and false negative (FN) rate as the probability threshold increases.

Fig. 3 shows the false positive and false negative rates as the probability threshold Th_{cp} increases from 0.5 to 0.95. The curves with solid label icons are the results by applying the refinement mentioned in Section 4.3 (with a variance threshold of 0.11), and the curves with hollow icons are the results without using the refinement. From the figure, we can see that both false positive rate and false negative rate are stable when $0.8 \leq Th_{cp} \leq 0.9$. By using the refinement algorithm, we largely reduce the false positive rate, while the false negative rate only has a slight increase.

The learned dependencies from the Hadoop logs correctly describe the control flow between system components. For example, a new task added in JobTracker causes a new task lunched by a TaskTracker. The more interesting thing is that some detected dependencies can form a dependency chain which can also help us to understand the dependency across several components. For example, a Reduce task starts its commit action, then the NameNode performs a data block allocation operation, and the DataNode carries out a HDFS_WRITE operation.

6 Application on Root Error Localization

In many distributed systems, an error occurring at one component often causes execution anomalies of other components due to inter-component dependencies. Therefore, a group of related errors from different system components are often detected at the same time. Figuring out the error propagation path and locating the root error site (not root cause) become an important step of problem diagnosis. In this section, we show an example to find the relationship among a set of related errors using the learned inter-component dependencies.

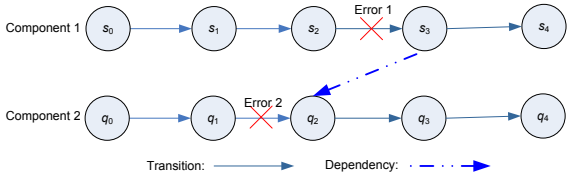

Figure 4. An example of relate errors.

6.1 The Basic Idea

The basic idea of error localization is illustrated in Fig.4. In a normal execution of Component 1, Component 1 prints log keys from s_0 to s_4. Similarly, Component 2 prints log keys from q_0 to q_4. In an error case, we detect Error 1 and Error 2 in Component 1 and Component 2, since a transition from s_2 to s_3 and a transition from q_1 to q_2 cannot happen. We call s_3 and q_2 ***inaccessible*** log of Error1 and Error2 respectively. Because log item q_2 of Component2 is dependent on log item s_3 of Component1, Component2 cannot print out q_2 if Component1 does not print s_3. Therefore, given Error1 and Error2, we can conclude that Error1 and Error2 are related and that Error1 causes Error2 in maximum likelihood. This shows the simple and basic idea behind our root error localization approach. Due to space limitation, we do not address the details.

6.2 Case Study

JVM memory overflow is a known bug of Hadoop v0.19 whose number is HADOOP-4906 [17]. In order to easily reproduce the error, we select one slave machine and set the available memory of the JVM running on the machine to a small value e.g. 10M. During the execution, a task fails due to memory exhaustion and finally causes the Job-Tracker to reassign the task to another machine. Our error detection algorithm in [16] can detect two errors: one is that the task abnormally terminates with a log message of "???[1] done; removing files" in TaskTracker. The other is that JobTracker prints task failure information. Based on our learned dependencies, we can quickly find that the second error is caused by the first one. Although we can manually obtain this casual information with some basic knowledge of Hadoop work flow, our algorithm can quickly locates the root error without using the application specific knowledge.

7 Conclusion

Knowledge of dependences provides an essential basis for distributed system management tasks including fault localization and anomaly detection. In this paper, we propose a technique to automatically discover inter-component dependencies in a distributed system based on unstructured log analysis. We first separate log keys and parameters from free form log messages, and then discover dependencies by leveraging the co-occurrence of log keys and the correspondence analysis of parameters. Our technique requires neither additional system instrumentation nor any application specific knowledge. We even do not require that the time stamps of logs from different machines are precisely synchronized. Experimental results on Hadoop demonstrate the power of our proposed technique. To the best of our knowledge, this is the first attempt to learn inter-component dependencies from console logs in a black-box manner.

Future research directions include integrating co-occurrence, parameter correspondence and delay consistence into a uniform probabilistic framework, visualizing the results to give intuitive explanation for human operators, and applying to root error localization.

8 Reference

[1] P. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. A. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang, "Discovering Dependencies for Network Management", in Proc. of 5th Workshop on Hot Topics in Networks, 2006.

[2] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions", USENIX OSDI'08, 2008.

[3] M. Gupta, A. Neogi, M.K. Agarwal, and G. Kar, "Discovering Dynamic Dependencies in Enterprise Environments for Problem Determination", in Proc. of 14th Int. Workshop on Distributed systems: Operations and Management (DSOM), pp.221-233, Oct. 2003.

[4] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma, "Constellation: automated discovery of service and host dependencies in networked systems", TechReport, MSR-TR-2008-67, April, 2008.

[5] S. Kandula, R. Chandra, and D. Katabi, "What's Going On? Learning Communication Rules in Edge Networks", SIGCOMM'08, 2008.

[6] V. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, M. Zhang, "Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies", SIGCOMM'07, 2007.

[7] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal, "Semi-Automated Discovery of Application Session Structure", In Proc. of the 6th ACM conf. on Internet measurement, pp. 119-132, 2006.

[8] M. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, A. Sailer, "Problem Determination Using Dependency Graphs and Run-time Behavior Models", in Proc. of 15th Int. Workshop on Distributed Systems: Operations and Management (DSOM'04) , pp. 171-182, Nov. 2004.

[9] A. Simma, M. Goldszmidt, J. MacCormick, P. Barham, R. Black, R. Isaacs, and R. Mortier, "CT-NOR: representing and reasoning about events in continuous time", In UAI'08, July 2008.

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

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

[15] S. Ghemawat and S. Leung, "The Google File System", In ACM Symposium on Operating Systems Principles", Oct. 2003.

[16] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis", submitted to ICDM'09.

[17] Hadoop bug reporting portal. http://issues.apache.org/jira/browse/HADOOP

[1] The identifier of the map task attempt.