International Journal of Research, Science, Technology & Management
ISSN Online: 2455-2240, Volume 18 Issue 2, February 2020

Automated Anomaly and Root Cause Detection in Distributed Systems

Arpita Singhai, Roshni Malviya

***Abstract*— It is a challenging issue to identify a defective system in a large scale network. The data collected from the distributed systems for troubleshooting is very huge and may contain noisy data, so manual checking and detecting of the abnormal node is time consuming and error prone. In order to solve this, we need to develop an automated system that detects the anomaly. Though the defective node is found it has to be rectified, for this we need to know the root cause of the problem. In this paper we present an automated mechanism for node level anomaly detection in large-scale systems and the root cause for anomaly. A set of data mining techniques are used here to analyze the collected data and to identify the nodes acting differently from others. We use Independent Component Analysis (ICA) for feature extraction. We also present some of the mechanisms needed to know the root cause of the problem. So the results will be abnormal nodes and problem to be rectified in them. These can be validated manually.**

Keywords—node level anomaly identification, large-scale systems, data mining techniques, independent component analysis.

1 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. In the distributed systems every hour that a system is unavailable can cause undesirable loss of processing cycles, as well as substantial maintenance cost. When a system fails to function properly, health-related data are collected across the system for troubleshooting. Unfortunately, how to effectively find anomalies and their causes in the data has never been as straightforward as one would expect. Traditionally, human operators are responsible of examining the data with their experience and expertise. Such manual processing is time-consuming, error-prone, and even worse, not scalable. As the size and complexity of computer systems continue to grow, so does the need for automated anomaly identification. To address the problem, in this paper, we present an automated mechanism for node-level anomaly identification and the root cause for the anomaly behaviour of the node. By finding the abnormal nodes and the cause for the anomaly, system managers are able to know where to fix the problem and what problem to fix. The use of the node in the error state can lead to node failure. Hence, we seek to discover the nodes in error or failed states, which are also called abnormal states in the paper; we regard these nodes as anomalies that require further investigation to say what makes the node anomaly.

2 METHODOLOGY OVERVIEW

This automated mechanism can be triggered either periodically with a predefined frequency or by a system monitoring tool in case of unusual events. In this paper, we focus on detecting anomalies in homogeneous collection of nodes (also called “groups”),and the reason for their anomaly behaviour . The resulting list of anomalies and the causes will be sent to system administrators for final validation. As we will see later, by combining the fast processing capability of computers with human expertise, the proposed mechanism can quickly discover anomalies and the causes with a very high accuracy. The step we perform for anomaly detection are

2.1. Data transformation. It is collection of relevant data across the system and assembling them into a uniform format called feature matrix (generally in high dimensionality). Here, a feature is defined as any individually measurable variable of the node being observed, such as CPU utilization, available memory size, I/O, network traffic, etc.

2.2. Feature extraction. A feature extraction technique, such as ICA, is applied on the feature matrix to generate a matrix with much lower dimensionality,

2.3. Outlier detection. It determines the nodes that are “far away” from the majority as potential anomalies. By analyzing the low-dimensional

matrix produced by feature extraction, a cell-based algorithm is used to quickly identify the outliers.

2.4. Root cause detection: It determines whether a deadlock occurrence or memory leakage problem that caused the node to act like a anomaly.

3.IMPLEMENTATION DESCRIPTION

3.1. Data Collection: In this step we have to get the information of the nodes in order to start the working process. Here we use the remote method invocation of java to get information of each and every node to the server by remote procedural calls. We have to compare the collected data, as data is in arbitrary formats, data preprocessing has to be applied to convert it into a single format. Possible preprocessing includes converting variable-spaced time series to constant-spaced ones, filling in missing samplings, generating real-value samples from system logs, and removing period spikes or noises and normalization. The features to be collected are CPU usage, memory available, IO information etc.

3.2Feature Matrix Construction: the collected information must be constructed in the form of matrix with row representing a particular feature information and column representing particular node complete information.

Features	Description
1.CPU_SYSTEM_PROC1 2.CPU_SYSTEM_PROC2	Percentage of CPU utilization at system level
3.CPU_USER_PROC1 4.CPU_USER_PROC2	Percentage of CPU utilization at user level
5.CPU_USER_PROC1 6.CPU_USER_PROC2	Percentage of time CPU blocked for I/O
7.MEMORY_FREE	Amount of free memory (KB)
8.MEMORY_SWAPPED	Amount of virtual memory used(KB)
9.PAGE_IN	Page in from swap(KB/s)
10.PAGE_OUT	Page out to swap(KB/s)
11.IO_WRITE	No of blocks written per second
12.IO_READ	No of blocks read per second
13.CONTEXT_SWITCH	No of context switches per second.
14.PACKET_IN	No of packets received per second.
15.PACKET_OUT	No of packets transmitted per second.
16.COMP_PROC1 17.COMP_PROC2	Computation time
18.COMM_PROC1 19.COMM_PROC2	Communication time

3.3 Dimensionality Reduction Using ICA:

There are a number of algorithms for performing ICA. For the purpose of fast convergence, we choose the FastICA algorithm. Whitening is a typical pre-processing step used to simplify and reduce the complexity of ICA algorithms. It ensures that all the dimensions are treated equally before the algorithm is run. For a vector v, its whitening means that its covariance matrix is equal to the identity matrix, that is, $\frac{1}{n}vv^t = I$,where n is the number of nodes in the distributed system. To whiten the matrix F^n (feature matrix), we first calculate i.covariance matrix C , and

ii. calculate nonzero Eigenvalues of C.

iii.Put them in a descent order:

$\lambda_1 \geq \lambda_2 \geq \ldots \lambda_r$ let v=diag(λ_1, λ_2, …, λ_r) and E=[e_1,e_2,…,e_r] where e_i is the Eigenvector corresponding to λ_i.

iv.The whitened data of F^n are defined as

$X = V^{-1/2}E^TF^n$; where X is a r × n matrix and $r \leq m \times k$.

After whitening, ICA projects the data point.$x_i \in IR^r$ into a data point

$y_i \in IR^s$ as $y_i = W^T x_i$. Where W is the matrix obtained after whitening.

The convergence of FastICA is good . In our experiments, generally only a few iterations are needed,and the total calculation time is less than 0.1 second.[1]

3.4 Outlier Detection:

This step is to identify a subset of nodes that are significantly dissimilar from the majority. In the field of data mining, these nodes are called outliers. CellBasedAlgorithm is used for this purpose.

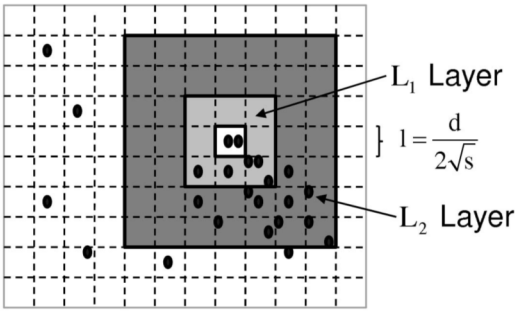


Fig. 2. Cell-based outlier detection.

The data space is partitioned into cells of length $l = d/2\sqrt{s}$. Each cell is surrounded by two layers: L1 (in the light-gray area) and L2 (in the dark-gray area).The cell-based algorithm works as follows. We first partition the data space that holds y={$y1, y2, y3, y4 \ldots, yn$}into cells of length

$l = d/2\sqrt{s}$ (see Fig. 2). Each cell is surrounded by two layers: 1) the first layer L1 includes those immediate neighbours and 2) the second layer L2 includes those additional cells within three cells of distance. For simplicity of discussion, let M be the maximum number of objects within the d-neighbourhood of an outlier (i.e., within a distance of d). According to the outlier definition, the fraction p is the minimum fraction of objects in the data set that must be outside the d-neighbourhood of an outlier.Hence,
$M = n(1-p)$. The cell-based algorithm aims to quickly identify a large number of outliers and nonoutliers .according to three properties, in the order as listed below:
1. If there are >M objects in one cell, none of the objects in this cell is an outlier.
2. If there are >M objects in one cell plus the L1 layer,none of the objects in this cell is an outlier.
3. If there are _M objects in one cell plus the L1 layer and the L2 layer, every object in this cell is an outlier.
These properties are used in the order to determine outliers and nonoutliers on a cell by cell basis rather than on an object by object basis. For cells not satisfying any of the properties,we have to resort to object by object processing.The above detection algorithm will separate the data set Y into two subsets: normal data set Yn and abnormal data set Ya. For each yi, we calculate its anomaly score:

$$\eta_{i=}\begin{cases}0 & , \quad Yi \epsilon Yn \\ d(Yi,\mu) & , \quad Yi \epsilon Ya\end{cases}$$

where μ is the nearest point belonging to the normal data set Yn. Anomaly score indicates the severity of anomaly.The abnormal subset, along with anomaly scores, will be
sent to system administrators for final validation.[2]

4.DETECTING AND SOLVING THE ROOT CAUSE

4.1 Insufficient CPU and Other CPU problems

4.1.1. Insufficient CPU

In the peak times of the work, CPU resources might be completely allocated and service time could be excessive too. In this situation, you must improve your system's processing ability. Alternatively, you could have too much idle time and the CPU might not be completely used up. In either case, you need to determine why so much time is spent waiting.

To determine why there is insufficient CPU, identify how your entire system is using CPU. Do not just rely on identifying how CPU is used by server processes. At the beginning of a workday, for example, the mail system may consume a large amount of available CPU while employees check their messages. Later in the day, the mail system may be much less of a bottleneck and its CPU use drops accordingly.

To address this CPU problem, we distinguish whether sufficient CPU resources are available and recognize when a system is consuming too many resources. Begin by determining the amount of CPU resources used by system when system is:

- Idle
- At average workloads
- At peak workloads

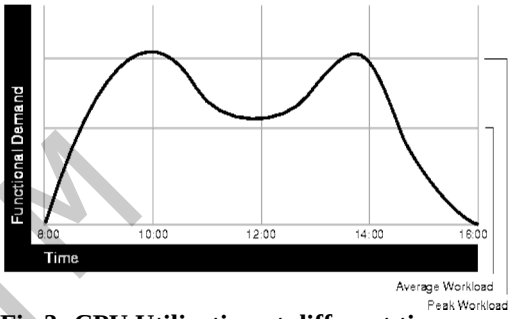


Fig 3: CPU Utilization at different times of working hours

The above figure shows the usage of 100 users working 8 hours a day, for a total of 800 hours per day. Each user entering one transaction every 5 minutes translates into 9,600 transactions daily. Over an 8-hour period, the system must support 1,200 transactions per hour, which is an average of 20 transactions per minute. If the demand rate were constant, you could build a system to meet this average workload.

However, usage patterns are not constant--and in this context, 20 transactions per minute can be understood as merely a minimum requirement. If the peak rate you need to achieve is 120 transactions per minute, you must configure a system that can support this peak workload.

For this example, assume that at peak workload server can use 90% of the CPU resource. For a period of average workload, then, server use no more than about 15% of the available CPU resource as illustrated in the following equation:

*20 tpm/120 tpm * 90% = 15%*

Where tpm is "transactions per minute".

If the system requires 50% of the CPU resource to achieve 20 tpm, then a problem exists: the system cannot achieve 120 transactions per minute using 90% of the CPU. However, if you tuned this system so it achieves 20 tpm using only 15% of the CPU, then, assuming linear scalability, the system might achieve 120 transactions per minute using 90% of the CPU resources.

As users are added to an application, the workload can rise to what had previously been peak levels. No further CPU capacity is then available for the new peak rate, which is actually higher than the previous.

Workload is a very important factor when evaluating your system's level of CPU use. During peak workload hours, 90% CPU use with 10% idle and waiting time may be understandable and acceptable; 30% utilization at a time of low workload may also be understandable. However, if your system shows high utilization at normal workloads, there is no more room for a "peak workload". You have a CPU problem if idle time and time waiting for I/O are both close to zero, or less than 5%, at a normal or low workload.

4.3 Detecting and Solving CPU Problems

4.3.1 Detection of System CPU Utilization

Commands such as **sar -u** on many UNIX-based systems enable you to examine the level of CPU utilization on your entire system. CPU utilization in UNIX is described in statistics that show user time, system time, idle time, and time waiting for I/O. A CPU problem exists if idle time and time waiting for I/O are both close to zero (less than 5%) at a normal or low workload.

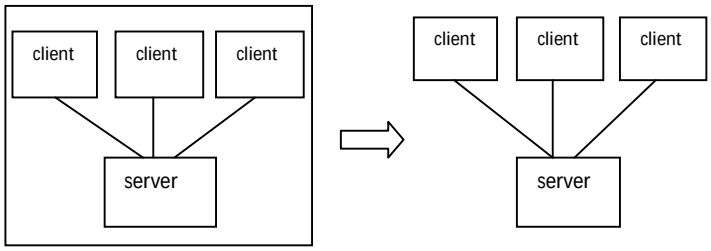
4.3.2Solving CPU Problems by Changing System Architectures

If you have maximized the CPU utilization power on your system and have exhausted all means of tuning your system's CPU use, consider redesigning your system on another architecture. Moving to a different architecture might improve CPU use. This section describes architectures you might consider using, such as:

- Single Tier to Two-Tier
- Multi-Tier: Using Smaller Client Machines
- Two-Tier to Three-Tier: Using a Transaction Processing Monitor
- Three-Tier: Using Multiple TP Monitors
- Oracle Parallel Server

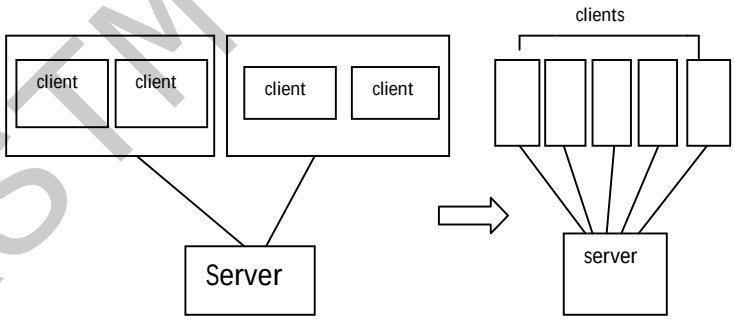
Single Tier to Two-Tier

Consider whether changing from several clients with one server, all running on a single machine (single tier), to a two-tier client/server configuration would relieve CPU problems.



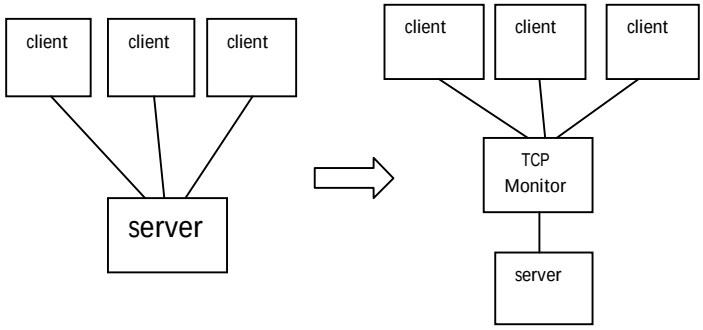
Multi-Tier: Using Smaller Client Machines

Consider whether using smaller clients improves CPU usage rather than using multiple clients on larger machines. This strategy may be helpful with either two-tier or three-tier configurations.



Two-Tier to Three-Tier: Using a Transaction Processing Monitor

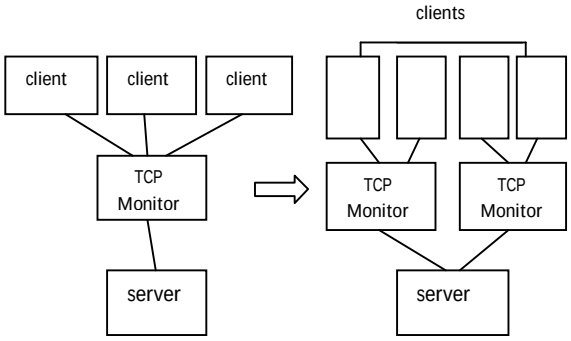
If your system runs with multiple layers, consider whether moving from a two-tier to three-tier configuration and introducing a transaction processing monitor might be a good solution.



Three-Tier: Using Multiple TP Monitors

Consider using multiple transaction processing monitors.

International Journal of Research, Science, Technology & Management
ISSN Online: 2455-2240, Volume 18 Issue 2, February 2020



4.4 Memory Management Problems and Detection

4.4.1Paging and Swapping

Use commands such as **sar** or **vmstat** to investigate the cause of paging and swapping problems.

4.4.2Memory Leakage

When the nodes in the distributed systems continuously utilize the memory and not leaving the memory then the state is considered as the memory leakage .it is a problem because the memory is being wasted.

4.4.3Memory leakage Detection

When a process tries to consume more memory than the virtual memory size, the system may crash. We use DMMA(Dynamic Memory Monitoring Agent), where we set the maximum memory consumption limit can be set virtually for each process. If memory utilization is higher than the maximum memory consumption limit DMMA consider the process in a bad state and identifies that process running on the node has memory leak. Otherwise DMMA considers the process in good state.

4.5 Dead Locks and their Detection in the Distributed Systems

4.5.1Deadlocks in Distributed Systems

Deadlocks in distributed systems are similar to deadlocks in single processor systems, only worse. They are harder to avoid, prevent or even detect. They are hard to cure when tracked down because all relevant information is scattered over many machines. People sometimes might classify deadlock into the following types: Communication deadlocks -- competing with buffers for send/receive Resources deadlocks -- exclusive access on I/O devices, files, locks, and other resources. We treat everything as resources, there we only have resources deadlocks. Four best-known strategies to handle deadlocks: The ostrich algorithm (ignore the problem) Detection (let deadlocks occur, detect them, and try to recover) Prevention (statically make deadlocks structurally impossible) Avoidance (avoid deadlocks by allocating resources carefully)

Distributed Deadlock Detection

Distributed Deadlock Detection Since preventing and avoiding deadlocks to happen is difficult, researchers works on detecting the occurrence of deadlocks in distributed system.Deadlock detection is realized by tracking which threads are waiting for which resources. When a cycle is detected, deadlock has occurred. Rather than tracking the waiting relation as an explicit graph, we use thread-local digests.

Let $T \in N$ represent threads and $R \in N$ represent resources. Further, we define owner : $R \rightarrow T$ to map resources to the threads which currently hold them.Thread T's digest, denoted DT , is the set of other threads upon which T is waiting, directly or indirectly.

The value of a given thread's digest depends on the thread's current state:

1. If thread T is not trying to acquire a resource,
   DT = {T}
2. If T is trying to acquire a resource R,
   DT = {T} ∪ Downer(R).

A thread trying to acquire a resource has a digest which includes itself as well as the digest of the resource's owner.

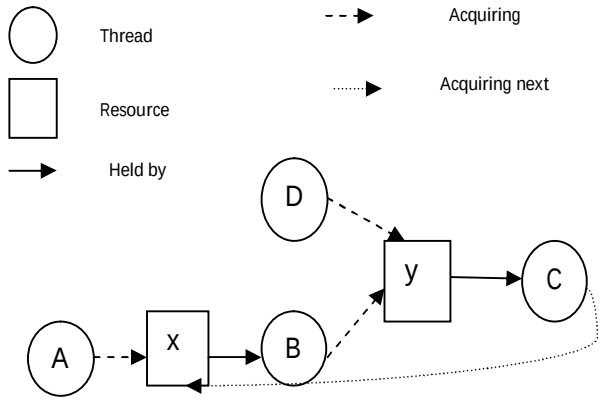


Fig5: Example of an explicit waits-for graph,of which Dreadlocks maintains small per-thread di-gests.

Moreover, the owner may itself be acquiring another resource and so the digest represents the transitive closure of the thread-centric waits-for graph. When a thread begins to acquire a resource

(moving from state 1 to state 2 above),it detects deadlock as follows:

Thread T detects deadlock when acquiring resource R if T ∈ Downer(R).

Consider the waits-for graph given in Figure , ignoring the dotted line for the moment. Thread A is attempting to acquire lock x held by thread B which, in turn, is trying to

acquire lock y held by thread C. Thread D is also trying to acquire lock y. Following the above rules, digests for this example are as follows:

DC = {C}
DD = {C, D}
DB = {C, B}
DA = {C, B, A}

The dotted line indicates that thread C tries to acquire lock x. It discovers itself in DB, detects a deadlock has been reached, and aborts. Digest Propagation Threads must propagate updates to digests to maintain per-thread transitive closures. Each lock must provide a field that references its owner's digest.

5.Conclusion

In this paper we have presented an automated mechanism for identifying anomalies in large scale systems. We have applied three techniques of data mining data transformation, feature extraction and outlier detection. The results show the abnormal nodes in large scale systems. In the abnormal nodes the root cause of the anomalies are identified. Finally these problems are manually validated.

References:

[1] A. Hyva¨rinen and E. Oja, Independent Component Analysis:Algorithms and Applications," Neural Networks, vol. 13, nos. 4/5,pp. 411-430, 2000.

[2] E. Knorr, R. Ng, and V. Tucakov, "Distance-Based Outliers: Algorithms and Applications," The VLDB J., vol. 8, no. 3, pp. 237- 253, 2000.

[3] Roohi Shabrin S., Devi Prasad B., Prabu D., Pallavi R. S., and Revathi P."*Memory Leak* Detection in *Distributed System"* - World Academy of Science, Engineering and Technology 16 2006. www.waset.org/journals/waset/v16/v16-15.pdf

[4] Eric Koskinen andMaurice Herlihy *Dreadlocks*: *Efficient Deadlock Detection* www.cl.cam.ac.uk/~ejk39/papers/dreadlocks-spaa08.pdf