ResearchGate

See discussions, stats, and author profiles for this publication at: http://www.researchgate.net/publication/275336864

Node Anomaly Detection for Homogeneous Distributed Environments

ARTICLE *in* EXPERT SYSTEMS WITH APPLICATIONS · MAY 2015

Impact Factor: 1.97 · DOI: 10.1016/j.eswa.2015.04.037

4 AUTHORS, INCLUDING:

Jian Xu
Nanjing University of Science and Technology
10 PUBLICATIONS **13** CITATIONS
SEE PROFILE

Yexi Jiang
Facebook
19 PUBLICATIONS **51** CITATIONS
SEE PROFILE

Chunqiu Zeng
Florida International University
10 PUBLICATIONS **14** CITATIONS
SEE PROFILE

Available from: Yexi Jiang
Retrieved on: 01 September 2015

Node Anomaly Detection for Homogeneous Distributed Environments

Jian Xu[1], Yexi Jiang[2], Chunqiu Zeng[2], Tao Li[2]

1. School of Computer Science and Technology, Nanjing University of Science and Technology, Nanjing, China.
   Email: dolphin.xu@njust.edu.cn
2. School of Computer Science, Florida International University, Miami, FL, USA.
   Email: {yjian004,czeng001,taoli}@cs.fiu.edu

Abstract: Identifying the anomalies is a critical task to maintain the uptime of the monitored distributed systems. For this reason, the trace data collected from real time monitors are often provided in form of streams for anomaly detection. Due to the dramatic increase of the scale of modern distributed systems, it is challenging to effectively and efficiently discover the anomalies from a voluminous amount of noisy and high-dimensional data streams. Moreover, the evolving of the system infrastructures brings new anomaly types that cannot be generalized as existing ones, making the existing anomaly detection solutions unavailable.

To address these issues, in this paper, we introduce a new type of anomalies called contextual collective anomaly. Then we propose a framework to discover this type of anomaly over a collection of data streams in real time. A primary advantage of this solution is that it can accurately identify the anomalies by taking both the contextual information and the historical information of a data stream into consideration. Also, the proposed framework is designed in a way with a low computational cost, and is able to handle large-scale data streams. To demonstrate the effectiveness and efficiency of our proposed framework, we empirically validate it on a real world cluster.

Keywords: Anomaly detection; multiple data streams; contextual anomaly; collective anomaly

1. Introduction

A homogeneous distributed environment generally consists of multiple computing nodes with the same hardware configuration, software environment and similar workloads. A typical example of the homogeneous distributed environment is the load-balanced system, which is widely used at the backend by the popular large-scale web sites like Amazon, Google and Facebook. In such a distributed environment, the computing nodes in a distributed system would behave similar to each other in the ideal situation (no anomaly and no occasional fluctuation). In such a situation, the observations (in terms of monitored metrics) of the nodes should be close to each other at any time. In practice, node anomaly might be caused by a variety of reasons, such as software aging, resource contention, and hardware failure, making the affected nodes behave differently from other nodes (Grottke & Trivedi, 2007). Overtime, a system is becoming more instable and it would fail to function properly due to the existence of anomaly nodes. Although the health-related data are collected across the system for troubleshooting, unfortunately how to effectively and efficiently identify anomalies and their root causes in the data has never been as straightforward as one would expect.

Traditionally, domain experts are responsible for examining the data with their experience and expertise. Such a manual process is time-consuming, error-prone, and even worse, not scalable. Due to the data scale and complexity, event the domain experts cannot fully identify the true anomalies and may also missing some deeply hidden anomalies. Moreover, as the behaviors of the distributed environment are likely changing over time, such temporal dynamics is difficult to be captured by the domain experts, as they may not be able to refresh their knowledge quick enough.

As the size and complexity of computer systems continue to grow, the difficulty for automated anomaly identification increases dramatically and it have far beyond the processing capability of the domain experts. The traditional expert systems that encoded the rules of the domain experts can only partially addressed the data scale problem. However, they cannot solve the complexity problem. This is because a distributed environment is dynamic. It is not likely such changing behaviors can be well captured by the static rules. Overtime, the deployed expert system based anomaly detection would gradually be outdated, as the rate of false positive and false negative would increase.

There are quite a few data processing and anomaly analysis infrastructures to enable automated anomaly identification. However, these existing data processing infrastructures are designed based on inherent non-stream programming paradigm such as Map/Reduce (Dean & Ghemawat, 2008), Bulk Synchronous Parallel (BSP) (Valiant, 1990), and their variations. To reduce the processing delay, these applications have gradually migrated to stream processing engines (Arasu et al., 2003; Chandrasekaran et al., 2003). As the infrastructures have been changed, anomalies in these applications are required to be identified online across multiple data streams. The new data characteristics and analysis requirements make existing anomaly detection solutions no longer suitable.

To address the problem, in this paper, we present a real time mechanism for node anomaly detection by taking both the node context information and the node historical information into consideration from multiple data streams.

1.1 A Motivating Example

Figure 1 illustrates the scenario of monitoring a 6-node computer cluster, where the x-axis denotes the time and the y-axis denotes the CPU utilization. The cluster has been monitored during time [0, t6]. At time t2, a computing task has been submitted to the cluster and the cluster finishes this task at time t4. As shown in Figure 1, two nodes (marked in dashed line) behave differently from the majority during some specific time periods. Node 1 has a high CPU utilization during $[t_1, t_2]$ and a low CPU utilization during $[t_3, t_4]$ while node 2 has a medium CPU utilization all the time. These two nodes with their associated abnormal periods are regarded as anomalies. Besides these two obvious anomalies, there is a slight delay on node 3 due to the network delay and a transient fluctuation on node 4 due to some random factors. However, they are normal phenomena in distributed systems and are not regarded as anomalies.

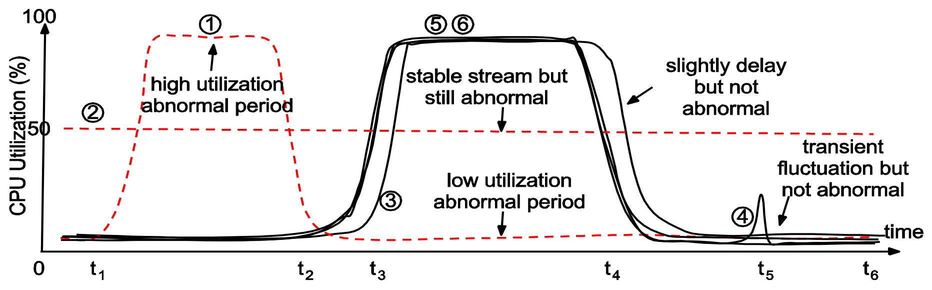


Figure 1 CPU utilization of a computing cluster

A quick solution for stream based anomaly detection is to leverage the techniques of complex event processing (CEP) [3, 4] by expressing the anomalies detection rules with corresponding continuous query statements. This rule-based detection method can be applied to the scenarios where the anomaly can be clearly defined. Besides using CEP, several stream based anomaly detection algorithms have also been proposed. They either focus on identifying the contextual anomaly over a collection of stable streams (Bu et al., 2009) or the collective anomaly from one stream (Anguilli & Fassetti, 2007; Pokrajac, Lazarevic, & Latecki, 2007). These existing methods are useful in many applications but they still cannot identify certain types of anomalies.

Figure 2 plots the ground truth as well as all the anomalies identified by existing methods including the CEP query with three different rules (Rule-CQ1, 2, and 3), the collective based anomaly detection (Breunig et al., 2000), and contextual based anomaly detection (Chandola, Banerjee, & Kumar, 2009).

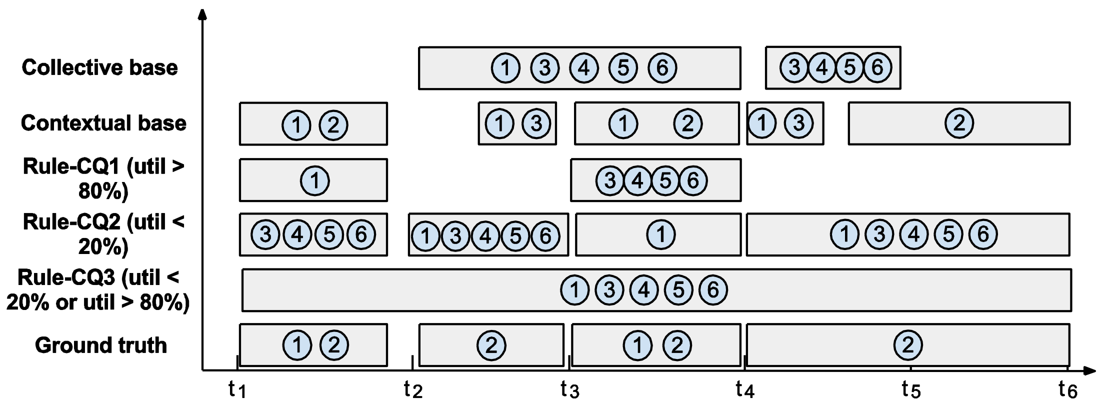


Figure 2 Identified anomalies in the motivating example (The box lists the IDs of abnormal streams during specified time period)

To detect the anomalies via CEP query, the idea is to capture the events when the CPU utilizations of nodes are too high or too low. An example query following the syntax of (Agrawal et al., 2008) can be written as follows:

PATTERN SEQ(Observation o[])

WHERE avg(o[].cpu) oper threshold (AND|OR avg(o[].cpu) oper threshold)*

WITHIN {length of sliding window}

where the selection condition in ***WHERE*** clause is the conjunction of one or more boolean expressions, ***oper*** is one of f>, <, <>, ==g, and **threshold** can be replaced by any valid expression. However, CEP queries are unable to correctly identify the anomalies in Figure 1 no matter how the selection conditions are specified. For instance, setting the condition as ***avg*****(o[].cpu) > {threshold}** would miss the anomalies during $[t_3, t_4]$ (Rule-CQ1); setting the condition as

***avg*(o[].cpu) < {threshold}** would miss the anomalies during $[t_1, t_2]$ (Rule-CQ2); and combining the two above expressions with OR still does not work (Rule-CQ3). Besides deciding the selection condition, how to rule out the situations of slight delays and transient fluctuations, and how to set the length of the sliding windows are all difficult problems when writing the continuous queries. The main reason is that the continuous query statement is not suitable to capture the contextual information where the "normal" behaviors are also dynamic (the utilizations of normal nodes also change over time in Figure 1).

Compared with CEP based methods, contextual anomaly detection methods (such as Gupta et al., 2013; Jiang, Chen, & Yoshihira, 2006) achieve a better accuracy as they utilize the contextual information of all the streams. However, one limitation of contextual based methods is that they do not leverage the temporal information of streams and are not suitable for anomaly detection in dynamic environments.

Therefore, these methods would wrongly identify the slightly delayed and fluctuated nodes as anomalies. For the given example, collective anomaly detection methods do not work well neither. This is because these methods would identify the anomaly of each stream based on its normal behaviors. Once the current behavior of a stream is different from its normal behaviors (identified based on historical data), it is considered as abnormal. In the example, when the cluster works on the task during time period $[t_3, t_4]$, all the working nodes would be identified as abnormal due to the sudden burst.

1.2 Contributions

In this paper, we propose an efficient solution to identify this special type of anomaly in the above example, named contextual collective anomaly. Contextual collective anomalies bear the characteristics of both contextual anomalies and collective anomalies. This type of anomaly is common in homogeneous distributed system monitoring, where data come from distributed but homogeneous data sources. We will formally define this type of anomaly in Section 2.

Besides proposing an algorithm to discover the contextual collective anomalies over a collection of data streams, we also consider the scale-out ability of our solution and develop a distributed streaming processing framework for contextual collective anomaly detection. More concretely, our contributions can be described as follows:

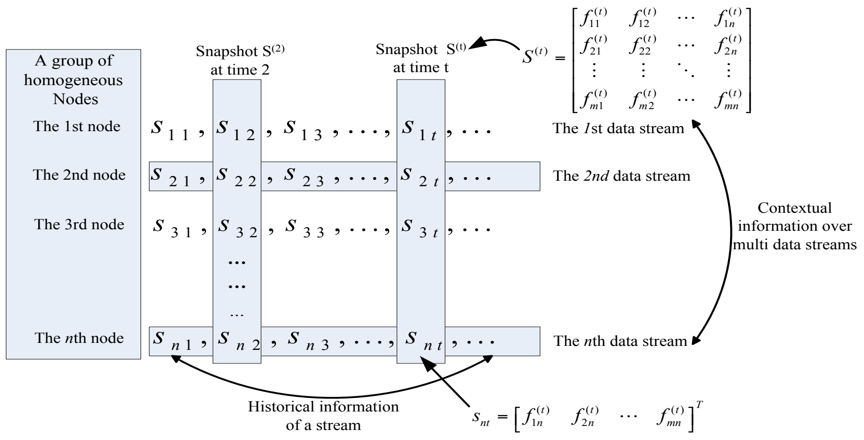
- We discuss the existing work of anomaly detection, especially streaming anomaly detection, and explain why the current types of anomalies cannot fully cover all the situations and why we need a new anomaly detection framework.
- We provide the definition of contextual collective anomaly and propose an incremental algorithm to discover the contextual collective anomalies in real time. The proposed algorithm combines the contextual as well as the historical information to effectively identify the anomalies.
- We propose a flexible three-stage framework to discover such anomalies from multiple data streams. This framework is designed to be distributed and can be used to handle large scale data by scaling out the computing resources. Moreover, each component in the framework is pluggable and can be replaced if a better solution is proposed in the future.
- We empirically demonstrate the effectiveness and efficiency of our solution through the real world scenario experiments and show that our solution has a better accuracy when system appears CPU-related faults, or IO-related faults, or a combinatory of these faults.

1.3 The Paper Outline

The rest of the paper is organized as follows. Section 2 gives a definition of contextual collective anomaly and then presents the problem statement. Section 3 provides an overview of our proposed anomaly detection framework. We introduce the three-stage anomaly detection algorithm in detail in Section 4. Section 5 presents the result of experimental evaluation. The related works are discussed in Section 6. Finally, we conclude in Section 7.

2. The Problem Statement

In this section, we first give the notations and definitions that are relevant to the anomaly detection problem and make it clear through Figure 3. Then, we formally define the problem based on the given notations and definitions.



A group of homogeneous Nodes
The 1st node
The 2nd node
The 3rd node
The nth node
Snapshot S(2) at time 2
Snapshot S(t) at time t
$S^{(t)} = \begin{bmatrix} f_{11}^{(t)} & f_{12}^{(t)} & \cdots & f_{1n}^{(t)} \\ f_{21}^{(t)} & f_{22}^{(t)} & \cdots & f_{2n}^{(t)} \\ \vdots & \vdots & \ddots & \vdots \\ f_{m1}^{(t)} & f_{m2}^{(t)} & \cdots & f_{mn}^{(t)} \end{bmatrix}$
$s_{11}, s_{12}, s_{13}, \ldots, s_{1t}, \ldots$
$s_{21}, s_{22}, s_{23}, \ldots, s_{2t}, \ldots$
$s_{31}, s_{32}, s_{33}, \ldots, s_{3t}, \ldots$
$s_{n1}, s_{n2}, s_{n3}, \ldots, s_{nt}, \ldots$
The 1st data stream
The 2nd data stream
The nth data stream
Contextual information over multi data streams
Historical information of a stream
$s_{nt} = \left[f_{1n}^{(t)} \quad f_{2n}^{(t)} \quad \cdots \quad f_{mn}^{(t)} \right]^T$

Figure 3 Data stream, stream collection and snapshot

Definition 1. Data Stream. Given a homogeneous distributed environment consisting of n computing nodes, for any a node $i, 1 \le i \le n$, a data stream of this node, S_i, is an ordered infinite sequence of data instances $\{s_{i1}, s_{i2}, s_{i3}, \ldots\}$. Each data instance s_{it} is the observation of data stream S_i at timestamp t and arrives at a fixed time interval, where $s_{it} = \left[f_{1i}^{(t)} \quad f_{2i}^{(t)} \quad L \quad f_{mi}^{(t)} \right]^T \in R^m$, and m is the number of features that are defined as any individually measurable variables of the node being observed, such as CPU utilization, available memory size, I/O, network traffic, etc. Note that all the observations have the same dimension.

A fault typically induces changes in multiple subsystems of a node, such as CPU, memory, I/O and network. For example, a memory leak may affect the amount of free memory and the CPU utilization rate; an operation to a malfunctioning disk may lead to huge IO time and long CPU idle time. Hence, in order to cover a broad fault space, it is necessary to collect and store feature data from all the subsystems per node. Furthermore, it is beneficial to track and store the tendencies of these features by collecting multiple samples. For the remaining paper, the terms "data instance" and "observation" would be used interchangeably. To make the notation uncluttered, we use s_i in places where the absence of timestamp do not cause the ambiguity.

Definition 2. Stream Collection. A stream collection $S = \{S_1, S_2, \ldots, S_n\}$ is a collection of the corresponding data streams from n computing nodes.

The input of our anomaly detection framework is a stream collection. For instance, in the motivating example, the input stream collection is $S = \{①,②,③,④,⑤,⑥\}$.

Definition 3. Snapshot. A snapshot is a matrix $S^{(t)} = [s_{1t} \quad s_{2t} \quad \ldots \quad s_{nt}] = \begin{bmatrix} f_{11}^{(t)} & f_{12}^{(t)} & \mathrm{L} & f_{1n}^{(t)} \\ f_{21}^{(t)} & f_{22}^{(t)} & \mathrm{L} & f_{2n}^{(t)} \\ \mathrm{M} & \mathrm{M} & \mathrm{O} & \mathrm{M} \\ f_{m1}^{(t)} & f_{m2}^{(t)} & \mathrm{L} & f_{mn}^{(t)} \end{bmatrix}$,

which captures the configuration of the stream collection *S* at time t and makes it easy to diagnose anomalies across different nodes.

Definition 4. Snapshot Anomaly Score. Given a snapshot $S^{(t)} = [s_{1t} \quad s_{2t} \quad \ldots \quad s_{nt}]$, its anomaly score is a vector $Q^{(t)} = [Q_{1t}, Q_{2t}, ..., Q_{it}, ..., Q_{nt}]^T$, where Q_{it} measures the amount of deviation of the observation s_{it} to the center of all the observations in snapshot $S^{(t)}$. To make the notation uncluttered, we use Q_i in places where the absence of timestamp do not cause the ambiguity. Snapshot Anomaly is a kind of contextual anomaly in nature. Under most of cases, an observation should be similar to the majority of the observations. The violation of this rule means that the stream may be a candidate of abnormal data streams. But, an observation of a data stream only reflects the transient behavior. Due to the appearance of transient fluctuation and slight phase shift, there is a high false-positive if we use a snapshot anomaly score only. To solve this issue, we take the historical information of a data stream into consideration to detect abnormal data streams. And we call this special type of anomaly as contextual collective anomaly.

Definition 5. Contextual Collective Anomaly. A contextual collective stream anomaly is denoted as a tuple $< S_i, [t_b, t_e], N_i >$, where S_i is the *i*-th data stream from the collection of data streams *S*, $[t_b, t_e]$is the associated time period when S_i is observed to constantly deviate from the majority streams in *S*, and $N_i = h(Q_i, I_i)$ indicates the severity of the anomaly, where Q_i denotes the snapshot anomaly score of data stream S_i, I_i measures the influence of the historical observation of data stream S_i and *h* is a function reflecting the idea of integrating the contextual information and the historical information.

In the motivating example, three contextual collective anomalies can be found in total. During time period $[t_1, t_2]$, node 1 behaves constantly different from the other nodes, so there is an anomaly $< 1, [t_1, t_2], N_1 >$. The other two contextual collective anomalies, $< 1, [t_3, t_4], N_1^{'} >$ and $< 2, [t_0, t_6], N_2 >$, can also be found with the same reason.

Problem Definition. The anomaly detection problem in our paper can be described below: Given a stream collection $S = \{S_1, S_2, ..., S_m\}$, identify the source of the contextual collective anomalies S_i, the associated time period $[t_b, t_e]$, as well as an anomaly quantification. Moreover, the detection has to be conducted on data streams that look-back is not allowed and the anomalies need to be identified in real time.

3. The System Framework

In this paper, we focus on detecting contextual collective anomalies in homogeneous collection of nodes (also called "groups"), and our proposed framework is shown as Figure 4. Our anomaly detection is based on two key observations. First, the nodes performing comparable activities generally exhibit similar behaviors (Tabatabaee & Hollingsworth, 2007; Mirgorodskiy, Maruyama, & Miller, 2006). Second, faults are rare events, so the majority of system nodes are functioning normally. For each group, three tightly coupled stages including the preprocessing stage, the scoring stage, and the alert stage, are applied to find the nodes that exhibit different behaviors from the majority. The functionality of the three stages is briefly described below.

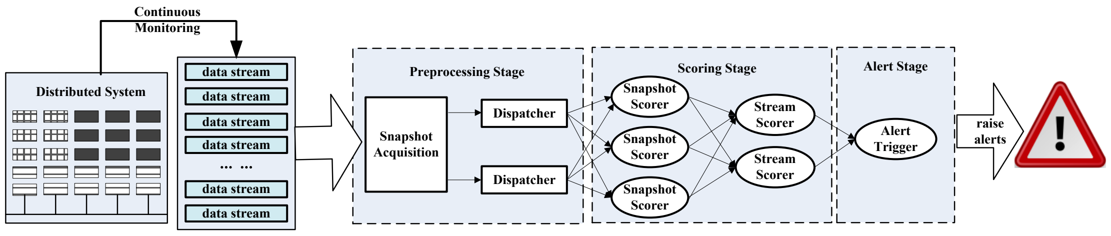


Figure 4 The distributed real time stream anomaly detection framework

- The Preprocessing stage.

On receiving the observations from external data sources, this stage is responsible for data stream preprocessing including common data preprocessing, snapshot acquisition, and snapshot dispatching. Possible common data 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. After common data preprocessing, data streams are assembled into snapshots. Finally, dispatchers are used to shuffle the snapshots to different downstream processing components.

- The Scoring stage

This stage quantifies the candidate anomalies using the snapshot scorer followed by the stream scorer. The snapshot scorer leverages contextual information to quantify the confidence of anomaly for each data instance at a given snapshot. Taking Figure 5 for example, it shows the data distribution by taking the snapshot of the 2-dimensional data instances of 500 streams at timestamp t. As shown, most of the data instances are close to each other and located in a dense area. These data instances are not likely to be identified as anomalies as their instance anomaly scores are small. On the contrary, a small portion of the data instances (those points that are far away from the dense region) have larger instance anomaly scores and are more likely to be abnormal.

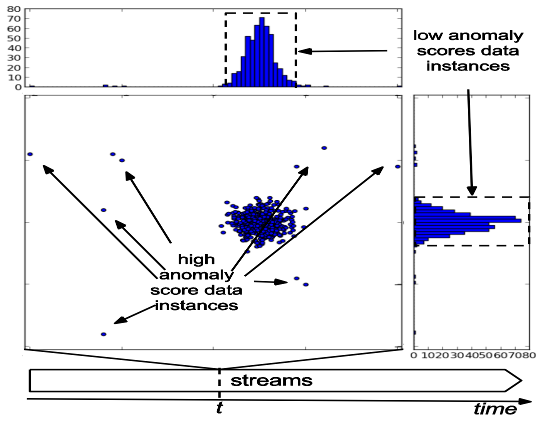


Figure 5 The snapshot at a certain timestamp

A data instance with a high anomaly score does not indisputably indicate its corresponding stream to be a real anomaly. This is because the transient fluctuation and phase shift are common in real world distributed environment. To mitigate such effects, the stream scorer is designed to handle the problem. In particular, the stream scorer combines the information obtained from the instance scorer and the historical information of each stream to quantify the anomaly confidence of each stream.

- The Alert stage.

The alert stage contains the alter trigger. The alert triggers leverage the unsupervised learning methods to identify and report the outliers. The advantage of our framework is reflected by the ease of integration, the flexibility, and the algorithm independence. Firstly, any external data sources can be easily fed to the framework for anomaly detection. Moreover, the components in every stage can be scaled-out to increase the processing capability if necessary. The number of components in each stage can be easily customized according to the data scale of different real applications. Furthermore, the algorithms in each stage can be replaced and upgraded with better alternatives and the replacement would not interfere with other stages.

To truly make the above mechanism useful in practice, we intend to provide two guarantees in the design of our anomaly identification system. First is the high accuracy, in terms of the low false alarm rate and extremely low missed rate (i.e., close to zero). Second is the time efficiency, meaning to quickly identify faulty nodes from the data collected, no matter how large are the data. The first guarantee is essential for the usefulness and effectiveness of the method, while the second guarantee is needed for quick detection and timely response.

4. Methodology

In this section, we present the details of our node anomaly detection mechanism.

4.1 Snapshot Acquisition & Dispatching

The goal of this step is to gather the system data for representing node behaviors, transform the data into a uniform format called as snapshot, and then dispatch them for data analysis.

The dispatching sub-step is an auxiliary stage in our framework. When the data scale (i.e., the number of streams) is too large for a single computing component to process, the dispatcher would shuffle the received observations to downstream computing components in the scoring stage. By leveraging random shuffling algorithm like Fisher-Yates shuffle (Fisher et al., 1949), dispatching can be conducted in constant time per observation. After dispatching, each downstream component would conduct scoring independently on sampled stream observations with identical distribution.

4.2 Snapshot Anomaly Quantification

Quantifying the anomaly in a snapshot is the first task in the scoring stage and we leverage snapshot scorer in this step. This score measures the amount of deviation of the specified observation sit to the center of all the observations in a snapshot at timestamp t.

A common solution to quantify the anomaly score of multi-dimension data is Local Outlier Factor (LOF) (Breunig et al., 2000). In principle, LOF measures the anomaly score using density-based clustering. This method is useful for online mining but is not suitable in our scenario due to the following two limitations: (1) LOF is not aware of the scale inconsistency among different dimensions. For dimensions with inconsistent scales, LOF would be dominated by the dimensions with large scales. Taking the system monitoring application for example, the CPU utilization is represented by the percentage of clock ticks used per second, while the memory usage is represented by the amount of RAM in KB, MB or GB, for which the later has a much larger scale. When using LOF, the dimension of memory usage would dominate the anomaly score. (2) The time of computing LOF score of observations increases superlinearly (O(*n*log*n*) as the number of observations increases. This time complexity is acceptable for online detection but is prohibitive for real time detection.

To address the above two limitations, we propose a simple yet efficient method to quantify the data instance anomaly scores. The basic idea is that the anomaly score of an observation is quantified as the amount of uncertainty it brings to the snapshot $S^{(t)}$. As the observations in a snapshot follow the normal distribution, it is suitable to use the increase of entropy to measure the anomaly of an observation.

To quantify the anomaly score, two types of variance are needed: the variance and the leave-one-out variance, where the leave-one-out variance is the variance of the distribution when one specific data instance is not counted.

Algorithm 1 Snapshot Anomaly Quantification

```
1.  Input: Snapshot $S^{(t)} = \{S_i : s_{it} \mid S_i \in S\}$, for $s_{it} \in R^m$.
2.  Output: Snapshot anomaly scores $Q \in R^n$.
3.  create a $m \times n$ matrix $M = [s_{1t}, s_{2t}, .., s_{nt}]$
4.  conduct 0-1 normalization on row of M;
5.  initialize $X = [x_1, x_2, ..., x_k, ..., x_m] \leftarrow 0^m$ and $Q^{(t)} = [Q_{1t}, Q_{2t}, Q_{nt}] \leftarrow 0^n$;
6.  $o_{avg} \leftarrow (\mathrm{IE}(S(1)), \mathrm{IE}(S(2)), \mathrm{IE}(S(m)))^T$
7.  $M \leftarrow (s_{1t} - o_{avg}, s_{2t} - o_{avg}, ..., s_{nt} - o_{avg})$;
8.  for $k = 1$ to $m$ do
9.      $x_k = \|k\text{th rows of } M\|_2^2$
10. end for
11. for all $s_{it} \in S^{(t)}$ do
12.     calculate $Q_{it}$ according to Equation (1);
13. end for
14. conduct 0-1 normalization of Q;
15. return $Q$
```

A naive algorithm to quantify the anomaly scores requires quadratic time ($O(dn + dn^2)$). By reusing the intermediate results, we propose an improved algorithm with time complexity linear to the number of streams. The pseudo code of the proposed algorithm is shown in Algorithm 1. As illustrated, matrix M is used to store the distances between each dimension of the observations to the corresponding mean. Making use of *M*, the leave-one-out variance can be quickly calculated as $\sigma_{ik} = \frac{n\sigma_k - M_{ik}^2}{n-1}$, where σ_k

denotes the variance of dimension k and σ_{ik} denotes the leave-one-out variance of dimension k by excluding s_{it} . As the entropy of normal distribution is $H = \frac{1}{2}\ln(2\pi e\sigma^2)$, the increase of entropy for observation s_{it} at dimension k can be calculated as

$$d_k = H_k^{'} - H_k = \ln\frac{\sigma_{ik}}{\sigma_k} = \ln\frac{n(x_k - M_{ik}^2)}{(n-1)x_k} \tag{1}$$

Summing up all dimensions, the snapshot anomaly score of s_{it} is $Q_{it} = \sum_{k=1}^{m} d_k$ .Note that the computation implicitly ignores the correlation between dimensions. This is because if an observation is an outlier, the correlation effect would only deviate it further from other observations.

4.3 Stream Anomaly Quantification

As a stream is continuously evolving and its observations only reflect the transient behavior, snapshot anomaly score alone would result in a lot of false-positives due to the transient fluctuation and slight phase shift. To mitigate such situations, it is critical to quantify the stream anomaly by incorporating the historical information of the stream.

An intuitive way to solve this problem is to calculate the stream anomaly score from the recent historical instances stored in a sliding window. However, this solution has two obvious limitations: (1) It is hard to decide the window length. A long sliding window would miss the real anomaly while a short sliding window cannot rule out the false-positives. (2) It ignores the impact of observations that are not in the sliding window. The observation that is just popped out from the sliding window would immediately and totally lose its impact to the stream.

To well balance the history and the current observation, we use *stream anomaly score* N_i to quantify how significant a stream S_i behaves differently from the majority of the streams. To quantify N_i , we exploit the exponential decay function to control the influence depreciation. Supposing Δt is a fixed time gap between any two adjacent observations, *the influence of an observation* s_{it_x} *at timestamp* $t_{x+k} = t_x + k\Delta t$ can be expressed as $Q_{it_x}(t_{x+k}) = Q_{it_x} e^{-\lambda kt}$, where λ is a parameter to control the decay speed. In the experiment evaluation, we will discuss how this parameter affects the anomaly detection results.

To make the notation uncluttered, we use t_{x-i} *to denote the timestamp that is* $i\Delta t$ *ahead of current timestamp* t_x , i.e. $t_{x-i} = t_x - i\Delta t$. Summing up the influences of all the historical observations, the overall historical influence I_{it} for current timestamp t can be expressed as Equation (2).

$$\begin{aligned} I_{it_x} &= Q_{it_{x-1}}\left(t_x\right) + Q_{it_{x-2}}\left(t_x\right) + Q_{it_{x-3}}\left(t_x\right) + \ldots \\ &= Q_{it_{x-1}} e^{-\lambda} + Q_{it_{x-2}} e^{-2\lambda} + Q_{it_{x-3}} e^{-3\lambda} + \ldots \\ &= e^{-\lambda}\left(Q_{it_{x-1}} + e^{-\lambda}\left(Q_{it_{x-2}} + e^{-\lambda}\left(Q_{it_{x-3}} + \ldots \right.\right.\right. \\ &= e^{-\lambda}\left(Q_{it_{x-1}} + I_{it_{x-1}}\right). \end{aligned} \tag{2}$$

The stream anomaly score of stream S_i is the weighted summation of the data instance anomaly score of current observation Q_{it} and the overall historical influence, i.e.,

$$N_{it_x} = \alpha Q_{it_x} + (1-\alpha) I_{it_x} \tag{3}$$

where α denote the weight of anomaly score of current observation and in this paper we set $\alpha = 0.5$

that means we make no discrimination on the contextual information and the historical information of a stream. As is shown in Equation (2), the overall historical influence can be incrementally updated with cost $O(1)$ for both time and space complexity. Therefore, *stream anomaly scorer* can be efficiently computed.

Comparing to the transient fluctuation, the real anomaly is more durable. Figure 6 shows the situations of a transient fluctuation (in the left subfigure) and a real anomaly (in the right subfigure). In both situations, the stream behaves normally before timestamp t_x. For the left situation, a transient fluctuation occurs at timestamp t_{x+1}, and then the stream returns to normal at timestamp t_{x+2}. For the right situation, an anomaly begins at timestamp t_{x+1}, lasts for a while till timestamp t_{x+k}, and then the stream returns to normal afterwards. Based on Figure 6, we show two properties about the stream anomaly score. The properties of stream anomaly score make our framework insensitive to the transient fluctuation and effective to capture the real anomaly.

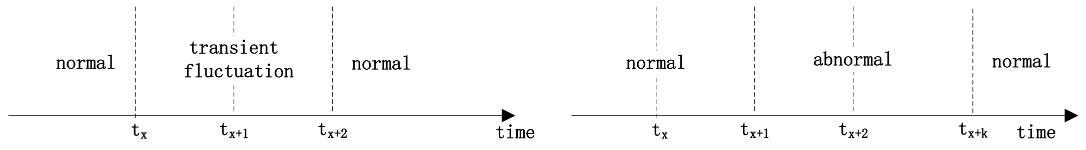


Figure 6 Transient Fluctuation and Anomaly

PROPERTY 1. *The increase of stream anomaly score caused by transient disturbance would decrease over time.*

PROOF. Suppose a transient fluctuation occurs at timestamp t_{x+1} in stream S_i, in the worst case, the difference between the data instance scores of the stream S_i and a non-fluctuated stream S_j is at most $d_{upper} = N_{it_{x+1}} - N_{jt_{x+1}} \leq \sqrt{m}$, where m is the number of dimensions. For simplicity, we use Euclidean distance to measure the difference between the data instance scores throughout this paper.

Let $\delta_1 = I_{it_{x+1}} - I_{jt_{x+1}}$. Before timestamp t_{x+1}, no stream is abnormal, so $I_{it_{x+1}}$ and $I_{jt_{x+1}}$ are close enough and the expectation $E(\delta_1) = 0$. At timestamp t_{x+1}, a transient fluctuation occurs in stream S_i. According to Equation (3), the difference of the stream anomaly scores is

$$\begin{aligned} N_{it_{x+1}} - N_{jt_{x+1}} &= \left(\alpha Q_{it_{x+1}} + (1-\alpha) I_{it_{x+1}}\right) - \left(\alpha Q_{jt_{x+1}} + (1-\alpha) I_{jt_{x+1}}\right) \\ &= \alpha(Q_{it_{x+1}} - Q_{jt_{x+1}}) + (1-\alpha)\delta_1 \\ &\approx \alpha(Q_{it_{x+1}} - Q_{jt_{x+1}}) \end{aligned} \tag{4}$$

At timestamp t_{x+2}, S_i turns to be normal again, so $Q_{it_{x+2}}$ equals to $Q_{jt_{x+2}}$ on average. Let $\delta_2 = Q_{it_{x+2}} - Q_{jt_{x+2}}$, we can also get $E(\delta_2) = 0$. Accordingly, the difference between the corresponding stream anomaly scores at timestamp t_{x+2} becomes

$$\begin{aligned}N_{it_{x+2}} - N_{jt_{x+2}} &= (\alpha Q_{it_{x+2}} + (1-\alpha)I_{it_{x+2}}) - (\alpha Q_{jt_{x+2}} + (1-\alpha)I_{jt_{x+2}}) \\ &= \alpha(Q_{it_{x+2}} - Q_{jt_{x+2}}) + (1-\alpha)(I_{it_{x+2}} - I_{jt_{x+2}}) \\ &= (1-\alpha)(I_{it_{x+2}} - I_{jt_{x+2}}) \\ &\approx (1\text{-}\alpha)(e^{-\lambda}(Q_{it_{x+1}} + I_{it_{x+1}}) - e^{-\lambda}(Q_{jt_{x+1}} + I_{jt_{x+1}})) \\ &= (1\text{-}\alpha)e^{-\lambda}(Q_{it_{x+1}} - Q_{jt_{x+1}} + \delta_1) \\ &< (1\text{-}\alpha)(Q_{it_{x+1}} - Q_{jt_{x+1}}) + (1-\alpha)\delta_1 \\ &\approx (1\text{-}\alpha)(Q_{it_{x+1}} - Q_{jt_{x+1}}) = \alpha(Q_{it_{x+1}} - Q_{jt_{x+1}}) = N_{it_{x+1}} - N_{jt_{x+1}}\end{aligned} \tag{5}$$

According to Equation (6), it is known that at timestamp t_{x+2}, the effect of fluctuation at timestamp t_{x+1} decreases. □

PROPERTY 2. *The increase of stream anomaly score caused by anomaly would be accumulated over time.*

PROOF. If a stream begins to be abnormal at timestamp t_{x+1}, its data instance scores would become larger than those of the normal streams during the abnormal period. Suppose ε is the difference of the data instance scores between the abnormal stream S_i and a normal stream S_j, at timestamp t_{x+1}. since both streams S_i and S_j are normal before timestamp t_{x+1}, we have $\delta_1 = I_{it_{x+1}} - I_{jt_{x+1}}$ and the expectation $E(\delta_1) = 0$. The difference of the stream anomaly scores is

$$\Delta = N_{it_{x+1}} - N_{jt_{x+1}} = \alpha(Q_{it_{x+1}} - Q_{jt_{x+1}}) + (1-\alpha)(I_{it_{x+1}} - I_{jt_{x+1}}) \approx \alpha(Q_{it_{x+1}} - Q_{jt_{x+1}}) = \varepsilon \tag{6}$$

At timestamp t_{x+2}, since the stream S_i is still in the abnormal period, the difference of data instance scores is still larger than or equal to ε, and the difference of stream anomaly scores between these two streams at timestamp t_{x+2} is

$$\begin{aligned}\Delta' = N_{it_{x+2}} - N_{it_{x+2}} &= \alpha(Q_{it_{x+2}} - Q_{jt_{x+2}}) + (1-\alpha)(I_{it_{x+2}} - I_{jt_{x+2}}) \\ &\geq \varepsilon + (1-\alpha)e^{-\lambda}((Q_{it_{x+1}} + I_{it_{x+1}}) - (Q_{jt_{x+1}} + I_{jt_{x+1}})) \\ &\approx \varepsilon + \alpha e^{-\lambda}(Q_{it_{x+1}} - Q_{jt_{x+1}}) \\ &= \varepsilon + e^{-\lambda}\varepsilon = (1 + e^{-\lambda})\varepsilon > \varepsilon.\end{aligned} \tag{7}$$

According to Equation (7), we can conclude that once a stream becomes abnormal, the difference between its stream anomaly score and those of the normal streams would increase over time. □

Similar properties can also be shown for the situation of slight shifts. A slight shift can be treated as two transient fluctuations occur at the beginning and the end of the shift. In the next section, we will leverage these two properties to effectively identify the anomalies in the ALERT STAGE.

4.4 Alert Triggering

Most of the stream anomaly detection solutions (Ge et al., 2010) identify the anomalies by picking the streams with top-k anomaly scores or the ones whose scores exceed a predefined threshold. However, these two approaches are not practical in real world applications for the following reasons: (1) Threshold is hard to set. It requires the users to understand the underlying mechanism of the application to correctly set the parameter. (2) The number of anomalies is changing all the time. It is possible that more than k anomaly streams exist at one time, then the top-k approach would miss these real anomalies.

To eliminate the parameters, we propose an unsupervised method to identify and quantify the anomalies by leveraging the distribution of the anomaly scores. The first step is to find the median of the stream anomaly scores N_{median}. If the distance between a stream anomaly score and the median score is larger than the distance between the median score and the minimal score $N_{\min}$, the corresponding stream is regarded as abnormal. As shown in Figure 7, this method implicitly defines a dynamic threshold (shown as the dashed line) based on the hypothesis that there is no anomaly. If there is no anomaly, the skewness of the anomaly score distribution should be small and the median score should be close to the mean score. If the hypothesis is true, $N_{median} - N_{\min}$ should be close to half of the distance between the minimum score and the maximum score. On the contrary, if a score N_i is larger than $2 \times (N_{median} - N_{\min})$, the hypothesis is violated and all the streams with scores at least N_i are abnormal.

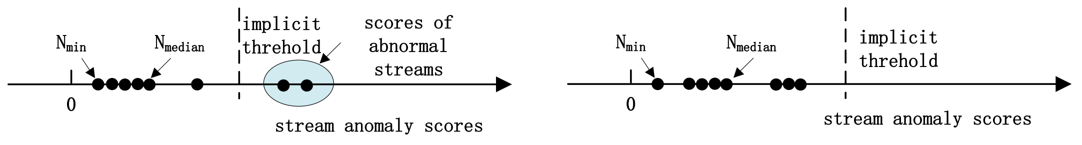


Figure 7 Abnormal Streams Identification

Besides the general case, we also need to handle one special case: a transient fluctuation occurs at the current timestamp. According to Property (1) in Section 4.3, the effect of transient fluctuation is at most d_{upper} and it will monotonically decrease. Therefore, even a stream whose anomaly score is larger than $2 \times (N_{median} - N_{\min})$, it can still be a normal stream if the difference between its anomaly score and $N_{\min}$ is smaller than d_{upper}. To prune the false-positive situations caused by transient fluctuation, the stream is instead identified as abnormal if

$$N_i > \max\left(2\left(N_{median} - N_{\min}\right), N_{\min} + d_{upper}\right) \tag{8}$$

Another thing needs to be noted is that the stream anomaly scores have an upper bound $\frac{d_{upper}}{1-e^{-\lambda}}$. According to the property of convergent sequence, the stream anomaly scores of all streams would converge to this upper bound. When the values of stream anomaly scores are close to the upper bound, they tend to be close to each other and hard to be distinguished. To handle this problem, we reset all the stream anomaly scores to 0 whenever one of them close to the upper bound.

In terms of the time complexity, the abnormal streams can be found in $O(n)$ time. Algorithm 2 illustrates the algorithm of stream anomalies identification. The median of the scores can be found in $O(n)$ in the worst case using the BFPRT algorithm (Blum et al., 1973). Besides finding the median, this algorithm also partially sorts the list by moving smaller scores before the median and larger scores after the median, making it trivial to identify the abnormal streams by only checking the streams appearing after the median.

Algorithm 2 Stream Anomaly Identification

```
1. INPUT: $\lambda$, and unordered stream profile list $S = \{S_1, ..., S_n\}$.
2. $mIdx \leftarrow \left\lceil \frac{n}{2} \right\rceil$
3. $N_{median} \leftarrow BFPRT(S, mIdx)$
4. $N_{\min} \leftarrow \min(S_i.score \mid 0 \le i \le mIdx)$
5. $N_{\max} \leftarrow N_{median}$
```

```
6.  for $i \leftarrow mIdx$ to $n$ do
7.    if Condition (4) is satisfied then
8.      Trigger alert for $S_i$ with score $N_i$ at current time.
9.      if $N_i > N_{\max}$ then
10.       $N_{\max} = N_i$
11.     end if
12.   end if
13. end for
14. if $N_{\max}$ is close to the upper bound then
15.     Reset all stream anomaly scores.
16. end if
```

5. Experimental Evaluation

We evaluate our prototype implementation on a computing cluster by manually injecting a variety of system faults. While experimental evaluation with real faults would be better, a major problem with this approach is that we do not often have the luxury of allowing systems to run for many days to see their behaviors. Further, it is not guaranteed that the system will experience a variety of faults during the experimental time. The generally accepted solution to this problem is to inject the effects of faults in a system and observe the behavior of the system under the injected faults (Barton et al., 1990). In our experiments, we are able to test the proposed anomaly detection algorithm on dozens of fault effects via fault injection, and show that our proposed framework can effectively and efficiently discover the abnormal behaviors of the computer nodes with high precision and low latency.

5.1 Experiment Settings

We use a Linux cluster at Nanjing University of Science & Technology (NUST) as our experiment platform, which consists of 76 computing nodes with 3.4GHz Intel I3 2 processors and connects each node via a Gigabit LAN. The cluster is running Unbtun 12.04 and Hadoop 0.20.2, and a parameter sweep application that is submitted on these nodes. The application solves dense linear equations by using Gaussian Elimination method, thereby performing comparable computation tasks.

We randomly inject faults into the experiment platform by generating faulty threads in the back end—separating from the application threads, and test whether different mechanisms can effectively identify the faulty nodes. Three random factors are considered in our fault injection. First is to decide how many nodes to inject faults, second is to determine which nodes to inject faults, and the last is to decide the type(s) of fault(s) to inject. In our experiments, less than 10 percent of all nodes are randomly selected for fault injection. The inclusion of the zero cases is to test our anomaly detection mechanism under a fault-free environment. Totally, four typical types of faults are tested as follows:

- Memory leaking: On randomly selected nodes, besides the normal computation threads, we introduce threads to generate memory leaking on the nodes, meaning that these threads continue consuming memory without releasing it periodically.
- Unterminated CPU-intensive threads: On randomly selected nodes, the injected threads compete for the CPU resource with the normal computation threads on the nodes.
- Frequent I/O operations: On randomly selected nodes, we introduce extra I/O intensive threads, which keep reading and writing a large number of bytes from local disks.
- Network volume overflow: On randomly selected nodes, additional threads are introduced to keep transferring a large number of packets among them.

The source code of injection program is available at https://github.com/yxjiang/system-noiser. We select these faults based on the literature and our own experience on system log analysis (Gujrati et al., 2007;Gu et al., 2008; Park et al., 2008). For example, we have found that some job hang failures are triggered by deadlock, some network-related failures like packet loss are caused by heavy traffic volume, and some node map file failures are due to frequent IO operations. In our experiments, these faults are supposed to originate from the system rather than from the application. They can affect multiple subsystems in a node, including memory, CPU, I/O, and network, and eventually lead to system failure and/or performance degradation.

To collect the health-related data across the computing cluster for troubleshooting, we leverage a distributed system monitoring tool (Xu & Xu, 2009) developed in our previous work to collect 14 features from CPU, memory, I/O, and network per node at the operating system layer. These features are summarized in Table 1.

TABLE 1 FEATURE LIST

No.	Features	Description	Category
1	CPU_USAGE_PROC	CPU utilization	CPU
2	CPU_CTXSWITCH	No. of context switches per second	
3	USER_TIME_PROC1	process time spent in user mode	
4	SYS_TIME_PROC	process time spent in system mode	
5	CPU_IO_PROC	Percentage of time CPU blocked for I/O	
6	FREE_MEM	Available memory (MB)	memory
7	SWAPPED_MEM	Virtual memory (MB)	
8	PAGE_IN	No. of virtual page paged in from swap per second	
9	PAGE_OUT	No. of virtual page paged out to swap per second	
10	MEM_FRAGMENT	No. of external memory fragmentation	
11	IO_READ	No. of disk read operations per second	IO
12	IO_WRITE	No. of disk write operations per second	
13	NIC_RXBYTE	No. of received bytes by the NIC device per second	Network
14	NIC_TXBYTE	No. of transmitted bytes by the NIC device per second	

Then we deploy the proposed anomaly detection program on an external computer to analyze the collected trace data in real time. To well evaluate our proposed framework, we terminate all the irrelevant processes running on these nodes.

5.2 Results

We conduct two sets of experiments: 1) single-fault tests, where one type of faults are injected into the system and 2) multi-fault tests, where multiple types of faults are injected into the system. For each experiment, we conduct 10 runs and the results shown here are the averages of multiple runs. Through these injections, we can answer the following questions about our framework: (1) whether our framework can identify the anomalies with different types of root causes; (2) whether our framework can identify multiple anomalies occurring simultaneously.

5.2.1 Single-fault Tests

In the first set of experiments, we inject one type of faults onto 0-7 randomly selected nodes and assess whether our detection mechanism can correctly identify these abnormal nodes. The details of the partial injections are listed in Table 2.

TABLE 2 LIST OF INJECTIONS

NO.	Time Period	Node	Description
1	[100,150]	2,23	Keep CPU utilization above 95
2		34,48	keep number of IO operations per second above 100

3	[150,250]	17	Keep memory usage at 70%
4		55	keep number of packets transferred per second above 15000
5	[300,400]	23	Keep CPU utilization above 95
6		69	Keep memory usage at 70%
7		48	keep number of IO operations per second above 100
8	[400,450]	17	Keep memory usage at 70%
9		34	keep number of IO operations per second above 100
10	[500,550]	48	keep number of IO operations per second above 100
11		69	Keep memory usage at 70%
12	[700,800]	17	Keep memory usage at 70%
13		23	Keep CPU utilization above 95
14		55	keep number of packets transferred per second above 15000
15	[800,850]	2	Keep CPU utilization above 95
16		69	Keep memory usage at 70%
17		34,48	keep number of IO operations per second above 100
18	[900,950]	2	Keep CPU utilization above 95
19		69	Keep memory usage at 70%

Figure 8 illustrates the results of this experiment by plotting the actual injections (top 7 sub-figures) as well as the captured alerts (the bottom subplots), where the x-axis represents the time and y-axis represents the idled CPU utilization, idle memory usage, the number of IO operations per second, and the number of packets transferred per second or the number of anomalies in each timestamp. We evaluate the framework from three aspects through carefully-designed injections.

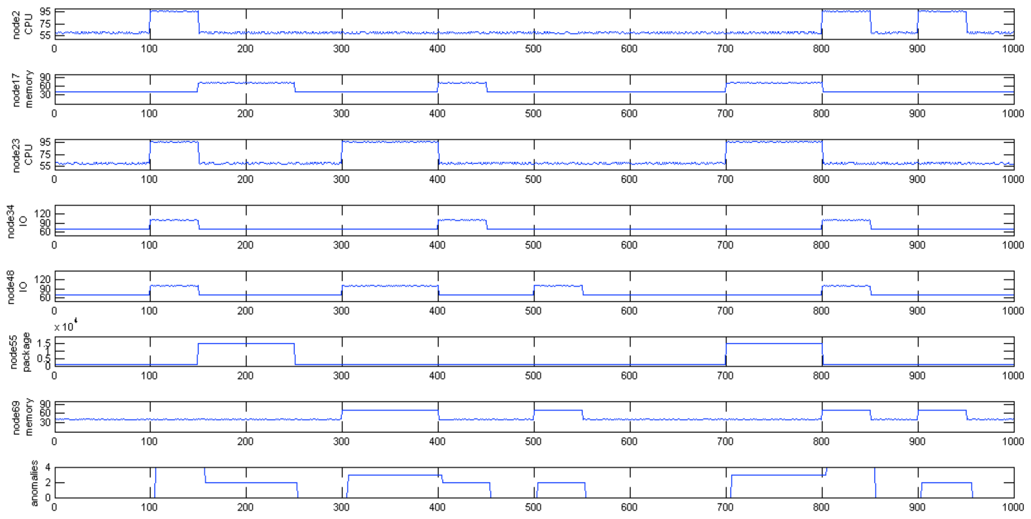


Figure 8 Single fault injections and the captured alerts

1. Single dimension (e.g. idle CPU utilization or idle memory usage) of a single stream behaves abnormally. This is the simplest type of anomalies. It is generated by injections No.5 and No.8 in Table 2. As shown in Figure 8, our framework effectively identifies these anomalies with the correct time periods.
2. Multiple streams behave abnormally simultaneously. This type of anomalies is generated by injection No.5, No.6 and No.8. During the injection time period, our framework correctly

identifies both anomalies (on node 23, node 48 and node 69).

3. Transient fluctuation and slight delay would not cause false-positive. As this experiment is conducted in a distributed environment, delays exist and vary for different nodes when executing the injections. Despite this intervention, our framework still does not report transient fluctuations and slight delays as anomalies.

Based on the evaluation results, we find that our solution is able to correctly identify all the anomalies in all these three different cases.

Further, we will delve into the details about the effectiveness and efficiency of our framework. To quantitatively measure the performance, we use F-measure to measure the accuracy and detection time delay to measure the efficiency. The precision and recall in computing F-measure are quantified according to the ground truth shown in Table 2. To investigate how λ affects the results, we conducted experiments with various λ values.

The experimental results of how λ affects the results accuracy is illustrated in Figure 9. To mitigate the randomness caused by the distributed environment, the precision, recall, and the F-measure are averaged with 10 runs. As shown, for the example of injecting CPU-related faults, as λ increases, precision increases but recall decreases. The reason for the decreasing of recall is as follows: The increase of λ causes the upper bound of stream anomaly scores to decrease and indirectly increases the reset frequency. After each reset of stream anomaly scores, some real anomalies would be skipped and they would reduce the recall. The result shows that the highest F-measure is 0.9351 while the lowest is 0.9032, which is stable. This is due to the changing of precision and recall cancels each other and makes F-measure insensitive to λ.This conclusion holds for other three cases. Last but not least, we also observe that detecting CPU-related faults has the highest accuracy, followed by IO-related faults, while detecting memory-related faults has the lowest accuracy, followed by network-related faults. We attribute this to the fact that CPU-related faults and IO-related faults can easily propagate throughout the system.

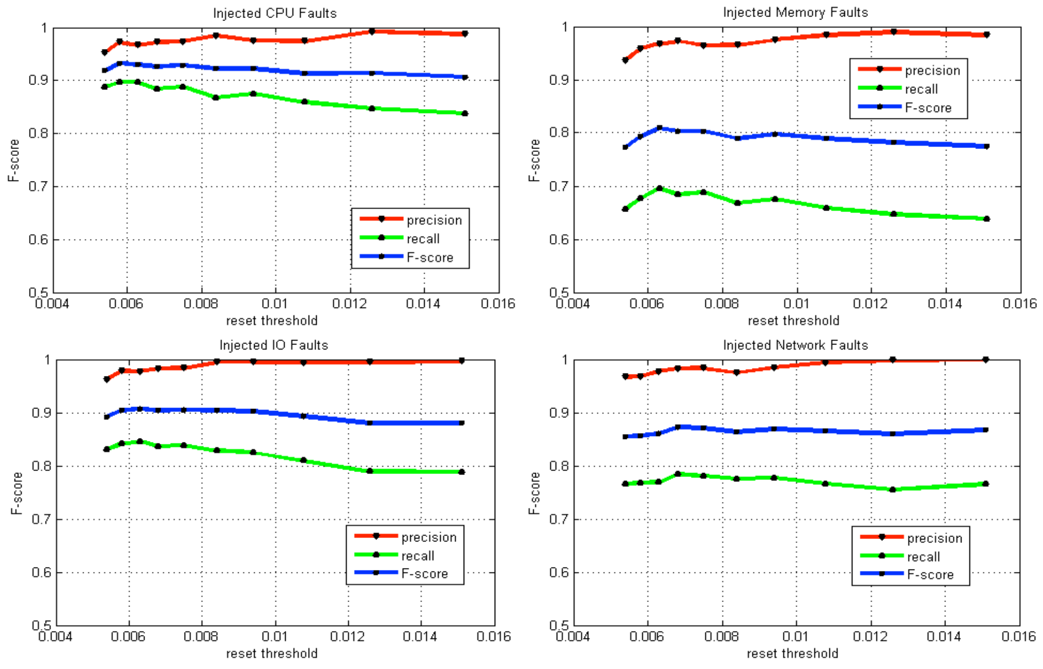


Figure 9 F-measure versus reset threshold

In terms of the time delay, our proposed framework is able to identify the anomalies in real time. As shown in Figure 10, the experimental results indicate that the average time delay in all the experiments is less than 6 seconds. We also notice that there is an obvious variance of the time delay due to the experiments that are conducted in a distributed system, where the environment is highly dynamic. Since the delay consists of network delay, injection execution delay, and the detection delay, the actual delay of our detection method should be less than the observed value.

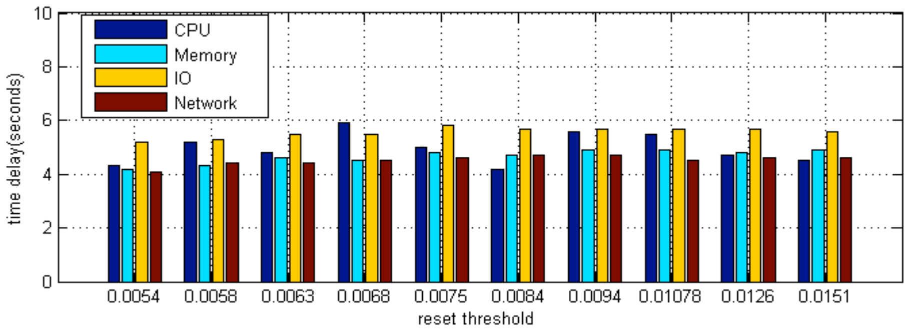


Figure 10 Time Delay versus reset threshold

5.2.2 Multi-fault Tests

In this set of experiments, different types of faults are simultaneously injected onto 0-3 nodes in the system. We have conducted experiments with two, three and four types of faults respectively. However, the results for the cases of three and four types of faults, are very similar to those for two-fault tests. Thus, we focus on discussion of the results for two-fault tests. The details of the injections are listed in Table 3.

TABLE 3 LIST OF INJECTIONS WITH MULTIPLE TYPES OF FAULTS

NO.	Time Period	Node	Description
1	[100,150]	2	Keep CPU utilization above 95 and keep number of packets transferred per second above 15000
2		48	keep number of IO operations per second above 100 and Keep memory usage at 70%
3	[300,400]	23	Keep CPU utilization above 95 and keep number of IO operations per second above 100
4		48	keep number of packets transferred per second above 15000 and Keep memory usage at 70%
5	[500,550]	2	Keep CPU utilization above 95 and keep number of IO operations per second above 100
6		48	keep number of IO operations per second above 100 and Keep memory usage at 70%
7	[650,750]	2	Keep memory usage at 70% and keep number of packets transferred per second above 15000
8		23	keep number of IO operations per second above 100 and keep number of packets transferred per second above 15000
9	[700,800]	23	Keep CPU utilization above 95 and keep number of packets transferred per second above 15000
10	[800,850]	48	Keep CPU utilization above 95 and Keep memory usage at 70%

Similar to Figure 8, Figure 11 illustrates the results of this experiment by plotting the actual injections (top 11 sub-figures) as well as the captured alerts (the bottom subplots). As shown in Figure 8, our framework effectively identifies these anomalies with the correct time periods under the case of injecting multiple faults simultaneously.

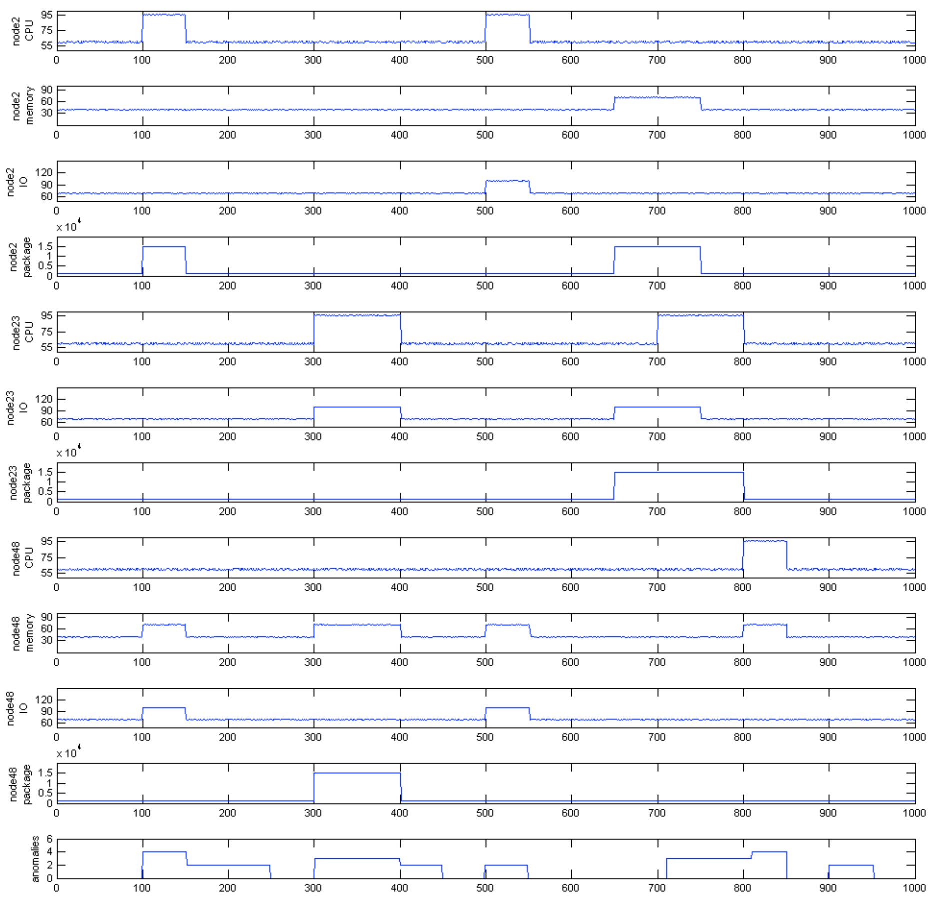


Figure 11 Multi-fault Injections and the captured alerts

As λ increases, precision increases but recall decreases. Thus, we will discuss the effectiveness and efficiency of our framework from the results with the worst recall for two-faults tests listed in the Table 4. Consistent with the conclusion that detecting CPU-related faults has the highest accuracy, followed by IO-related faults, while detecting memory-related faults has the lowest accuracy, followed by network-related faults draw from Figure 9, we can also find that there is a higher detection accuracy when injecting CPU-related faults or IO-related faults than the case of injecting memory-related faults or network-related faults. Moreover, the worst accuracy we got is 0.8878 in the case of injections of memory-related faults and network-related ones simultaneously. But this result is still better than both results, 0.7783 and 0.8764, got in the case of injecting single memory-related fault or single network-related fault respectively. Similarly, we can find that this conclusion always holds in the remaining cases, which shows that our detection solution can work well under complex runtime environments. In terms of the time delay, the experimental results indicate that the time delay in all the experiments is less than 5 seconds and less than the average time delay in case of injecting four single faults respectively, which shows that our proposed framework can also identify the anomalies in real time in case of multi-fault tests.

Table 4 Results in case of multi-fault tests

	Multiple Faults						Single Fault			
Types of Faults	**CPU & Mem.**	**CPU & IO**	**CPU & Net.**	**Mem. & IO**	**Mem. & Net.**	**IO & Net.**	**CPU**	**Mem.**	**IO**	**Net.**
F-Score	*0.9004*	*0.9467*	*0.9105*	*0.8999*	*0.8878*	*0.9073*	*0.9002*	*0.7783*	*0.8846*	*0.8764*

Avg. F-score	0.9087						0.8598			
Time Delay	4.2	4.2	4.2	4.3	4.3	4.3	4.3	4.7	5.3	4.4
Avg. Time Delay	4.25						4.675			

5.2.3 Comparison Analysis

To demonstrate the superiority of our framework, we also conduct experiments to identify the anomalies with the same injection settings described in Figure 11 using the alternative methods including contextual anomaly detection (CAD) and rule-based continuous query (Rule-CQ). The contextual anomaly detection is equivalent to the snapshot scoring in our framework. For the rule-based continuous query, we define four rules to capture three types of anomalies, including high CPU utilization (rule 1), high memory usage (rule 2), high number of IO operations per second (rule 3) and high number of network packets per second (rule 4) respectively. Different combinations of these rules are used in the experiments.

For Rule-CQ method, we experiment all the combinations: C1 (rule 1 or rule 2), C2 (rule 1 or rule 3), C3 (rule 1 or rule 4), C4 (rule 2 or rule 3), C5 (rule 2 or rule 4) and C6 (rule 3 or rule 3), and report the results. Table 5 quantitatively shows the precision, recall, and F-measure of these methods as well as the results of our method. The contextual anomaly detection method generates a lot of false alerts. This is because this method is sensitive to the transient fluctuation. Once an observation deviates from the others at a timestamp, an alert would be triggered. Similarly, the Rule-CQ method also generates many false alerts since it is difficult to use rules to cover all the anomaly situations. The low-precision and high-recall results of CAD and Rule-CQ indicate that all these method are too sensitive to fluctuations.

Table 5 Measures of different methods performed on data streams of the distributed environment

Measure **Algorithm**	**precision**	**Recall**	***F-measure***
CAD	0.4219	1.0000	*0.5935*
C1:Rule 1||2	0.3382	1.0000	*0.5055*
C2:Rule 1||3	0.5423	1.0000	*0.7032*
C3:Rule 1||4	0.4642	1.0000	*0.6340*
C4:Rule 2||3	0.3244	1.0000	*0.4899*
C5:Rule 2||4	0.1878	1.0000	*0.3154*
C6:Rule 3||4	0.2182	1.0000	*0.3582*
Our algorithm (worst case)	*0.9604*	*0.8623*	*0.9087*

6. Related Works

Leveraging machine learning and data mining techniques to facilitate the system management is always a hot research direction in both the communities of system and data mining (Zheng et al., 2014; Jiang et al., 2011). In this direction, automatic anomaly detection is an increasing popular topic that has drawn many interests from the researchers (Chandola, Banerjee, & Kumar 2009). Over the recent years, continuously efforts are paid towards this topic. Generally speaking, there are mainly two groups of related research directions: the model-based anomaly detection and the data-driven anomaly detection.

A model-based approach derives a probabilistic or analytical solution by modeling the system in a parameterized way. Hellerstein et al. proposed a model that is able to quantify the severity of anomaly in an unsupervised approach. This anomaly detection model is able to trigger the warnings when a deviation from the normal status learned by the model is detected (Hellerstein, Zhang, & Shahabuddin, 2001). Salehi et al. proposed an ensemble model based method to identify the anomalies in switching data streams (Salehi et al. 2014). Vaidynathan and Cross proposed an adaptive statistical data fitting method called MSET to enable the automatic anomaly detection using statistical testing (Vaidyanathan & Cross, 2003). Moreover, Hamerly and Elkan proposed a naive Bayesian-based model for disk failure prediction (Hamerly & Elkan, 2001) and Garg, Puliafito and Trivedi proposed a Semi-Markov reward model (Garg, Puliafito, & Trivedi, 1995), both focusing on failure detection using statistical or automation techniques. Although these proposed works are able to alleviate the management burden for system administrators, they ask the people to input model parameters, which is difficult for the people without solid mathematic or data mining background. Without properly setting of parameters, the model-based methods would have the difficulty of generating and maintaining an accurate model, especially given the unprecedented size and complexity of large-scale systems.

Besides the previously mentioned related work, recently, data mining and machine learning have received growing attention for failure diagnosis and prognosis. These methods extract fault patterns from system normal behaviors and detect abnormal observations based on the learned knowledge without assuming a priori model ahead of time. For example, in (Sahoo et al., 2003;Vilalta & Ma, 2002), the authors have presented several methods to predict failure events in IBM clusters. Fox et al. proposed an anomaly detection approach that considers both simple operational statistics and structural change in a complex distributed system (Fox, Kiciman, & Patterson, 2004). Fu and Xu (2007) have developed a framework called hPREFECTS for failure prediction in networked computing systems. Other representative studies include system log analysis (Oliner & Stearley, 2007; Schroeder & Gibson, 2006) and fault detection in syslogs (Stearley & Oliner, 2008).

Different from the model-based method, our approach belongs to the data-driven category (i.e., making decisions by gathering and analyzing large amounts of data), it focuses more on building a systematic framework for real time anomaly detection over data streams from large-scale systems (Jiang et al. 2014).

With the emerging requirements of mining data streams, several techniques have been proposed to handle the data incrementally (Jiang et al., 2013; Jiang et al., 2014; Liang et al., 2008). Pokrajac et al. (2007) modified the static Local Outlier Factor (LOF) method as an incremental algorithm, and then applied it to find data instance anomalies from the data stream. Takeuchi and Yamanishi (2006) trained a probabilistic model with an online discounting learning algorithm, and then use the training model to identify the data instance anomalies. Angiulli and Fassetti (2007) proposed a distance-based outlier detection algorithm to find the data instance anomalies over the data stream. Wu et al. (2014) proposed a data structure called RS-Forest for modeling the density anomalies over data streams. Pham et al (2014) proposed a residual space analysis based method to detect the anomalies in a large-scale data stream network. Liang et al. (2008) improved the efficiency of Lee's work by only computing the distances among the sub-trajectories in the same grid. As the aforementioned two algorithms require accessing the entire dataset, they cannot be adapted to trajectory streams. To address the limitation, Bu et al. (2009) proposed a novel framework to detect anomalies over continuous trajectory streams. They built local clusters for trajectories and leveraged efficient pruning strategies as well as indexing to reduce the computational cost. However, their approach identified anomalies based on the local-continuity property of the trajectory, while our method does not make such an assumption. Our approach is close to the work of (Ge et al.,2010), where they proposed an incremental approach to maintain the top-K evolving trajectories for traffic monitoring. However, their approach mainly focused on the geospatial data instances and ignored the temporal correlations, while our approach explicitly considers the temporal information of the data instances. Moreover, all the aforementioned works focused on the anomaly detection of a single stream, while our work is designed to discover the contextual collective anomalies over multiple data streams.

7. Conclusions

Anomaly detection is always a top priority for distributed system management and it draws the attention of many researchers in recent years. Due to the increasing of data scale and data complexity, existing anomaly detection methods gradually lose their abilities. To improve the effectiveness and efficiency of anomaly detection and to identify new types of anomaly, in this paper, we propose a real time anomaly detection framework to identify the contextual collective anomalies from a collection of streams. Our proposed method firstly quantifies the snapshot level anomaly of each stream based on the contextual information. Then the contextual information and the historical information are used in combination to quantify the anomaly severity of each stream. Based on the distribution of the stream anomaly scores, an implicit threshold is dynamically calculated and the alerts are triggered accordingly. To demonstrate the usefulness of the proposed framework, several sets of experiments are conducted to demonstrate its effectiveness and efficiency.

There are some future works we need to do to address the limitations of the current solution. The current proposed method can only be applied to the homogeneous distributed systems. In the real world, there are a lot of heterogeneous distributed systems that the behaviors of the nodes are not all the same. For example, the master node and the slave nodes would behave differently in the Hadoop distributed systems. In order to conduct the anomaly detection on this kind of distributed systems, a hybrid method that combines offline learning and online learning should be designed. For such a solution, the offline learning techniques would be used to learn the signatures of the normal behaviors of the system its history. Then the learned model would be applied to conduct the online learning for real time anomaly detection. Due to the highly dynamics of the distributed systems, this solution should be able to self-adaptive.

8. ACKNOWLEDGEMENT

The work was supported in part by the National Natural Science Foundation of China under grant number 61300053, the National Science Foundation under grants DBI-0850203, HRD-0833093, CNS-1126619, and IIS-1213026, the U.S. Department of Homeland Security under grant Award Number 2010-ST-06200039, Army Research Office under grant number W911NF-10-1-0366 andW911NF-12-1-0431, and an FIU Dissertation Year Fellowship.

9. References

J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman (2008). Efficient pattern matching over event streams. In Proceedings of SIGMOD.

F. Anguilli and F. Fassetti (2007). Detecting distance-based outliers in streams of data. In Proceedings of CIKM.

A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom (2003). STREAM: the stanford stream data manager. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data.

J. Barton, E. Czeck, Z. Segall, and D. Siewiorek (1990). "Fault Injection Experiments Using FIAT," IEEE Trans. Computers, vol. 39, no. 4.

M. Blum, R. Floyd, V.Pratt, R.Rivest, and R. Tarjan (1973). Time bounds for selection. Journal of Computer System Science.

M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander (2000). Lof: Identifying density-based local outliers. In SIGMOD.

Y. Bu, L. Chen, A. W.-C. Fu, and D. Liu (2009). Efficient anomaly monitoring over moving object trajectory streams. In Proceedings of KDD.

V. Chandola, A. Banerjee, and V. Kumar (2009). Anomaly detection: A survey. ACM Computing Surveys.

S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Honh, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah (2003). TelegraphCQ: continuous dataflow processing[C]//Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM.

J. Dean and S. Ghemawat (2008). MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

R. A. Fisher, F. Yates, et al (1949). Statistical tables for biological, agricultural and medical research. Statistical tables for biological, agricultural and medical research., (Ed. 3.).

A. Fox, E. Kiciman, and D. A. Patterson (2004). "Combining statistical monitoring and predictable recovery for self-management," in Proc. of WOSS, 2004.

S. Fu and C. Xu (2007). "Exploring Event Correlation for Failure Prediction in Coalitions of Clusters," Proc. Conf. Supercomputing (SC '07).

S. Garg, A. Puliafito, and K. Trivedi (1995). "Analysis of Software Rejuvenation Using Markov Regenerative Stochastic Petri Net," Proc. Sixth Int'l Symp. Software Reliability Eng.

Y. Ge, H. Xiong, Z.-H. Zhou, H. Ozdemir, J. Yu, and K. C. Lee (2010). Top-eye: Top-k evolving trajectory outlier detection. In Proceedings of CIKM.

M. Grottke and K. Trivedi (2007). "Fighting Bugs: Remove, Retry, Replicate and Rejuvenate," IEEE Computer, vol. 40, no. 2.

J. Gu, Z. Zheng, Z. Lan, J. White, E. Hocks, and B.H. Park (2008). "Dynamic Meta-Learning for Failure Prediction in Large-Scale Systems: A Case Study," Proc. Int'l Conf. Parallel Processing (ICPP).

P. Gujrati, Y. Li, Z. Lan, R. Thakur, and J. White (2007). "A Meta-Learning Failure Predictor for Blue Gene/L Systems," Proc. Int'l Conf. Parallel Processing (ICPP).

M. Gupta, A. B. Sharma, H. Chen, and G. Jiang (2013). Context-aware time series anomaly detection for complex systems. In WORKSHOP NOTES, page 14.

G. Hamerly and C. Elkan (2001), “Bayesian Approaches to Failure Prediction for Disk Drives,” Proc. Int’l Conf. Machine Learning (ICML).

J. Hellerstein, F. Zhang, and P. Shahabuddin (2001). “A Statistical Approach to Predictive Detection,” Computer Networks: The Int’l J. Computer and Telecomm. Networking, vol. 35,pp. 77-95.

G. Jiang, H. Chen, and K. Yoshihira (2006). Modeling and tracking of transaction flow dynamics for fault detection in complex systems. Dependable and Secure Computing, IEEE Transactions on, 3(4):312–326.

Y. Jiang, C.-S. Perng, T. Li, and R. Chang (2011). ASAP self-adaptive prediction system for instant cloud resource demand provisioning. In Proceedings of ICDM.

Y. Jiang, C.-S. Perng, T. Li, and R. Chang (2013). Cloud Analytics for Capacity Planning and Instant VM Provisioning. Network Management and System Management, IEEE Transactions on,10(3): 312–325.

Y. Jiang, C.-S. Perng, and T. Li (2014). META: Multi-resolution Framework for Event Summarization. SIAM International Conference on Data Mining.

Y. Jiang, C. Zeng, J. Xu, T. Li (2014). Real time contextual collective anomaly detection over multiple data streams. SIGKDD Workshop on Outlier Detection and Description under Data Diversity.

J.-G. Lee, J. Han, and X. Li (2008). Trajectory outlier detection: A partition-and-detect framework. In ICDE.

A. Mirgorodskiy, N. Maruyama, and B. Miller (2006). “Problem Diagnosis in Large-Scale Computing Environments,” In Proc. Conf. Supercomputing (SC).

B. Mozafari, K. Zeng, and C. Zaniolo (2012). High-performance complex event processing over xml streams. In SIGMOD.

A. Oliner and J. Stearley (2007). “What Supercomputers Say: A Study of Five System Logs,” Proc. Int’l Conf. Dependable Systems and Networks (DSN).

B. Park, Z. Zheng, Z. Lan, and A. Geist (2008). “Analyzing Failure Events on ORNL’s Cray XT4,” Proc. Conf. Supercomputing (SC ’08), (research poster).

D. Pham, S.Venkatesh, M. Lazarescu, S. Budhaditya (2014). Anomaly detection in large-scale data stream networks. Data Mining and Knowledge Discovery, 28(1), 145-189.

D. Pokrajac, A. Lazarevic, and L. J. Latecki (2007). Incremental local outlier detection for data streams. In Proceedings of CIDM.

R. Sahoo, A. Oliner, I. Rish, M. Gupta, J. Moreira, S. Ma, R.Vilalta, and A. Sivasubramaniam (2003). “Critical Event Prediction for Proactive Management in Large-Scale Computer Clusters,” In Proc. of ACM Special Interest Group on Knowledge Discovery in Data SIGKDD.

M. Salehi, C. A. Leckie, M. Moshtaghi, T. Vaithianathan (2014). A Relevance Weighted Ensemble Model for Anomaly Detection in Switching Data Streams. Advances in Knowledge Discovery and Data Mining.

B. Schroeder and G. Gibson (2006), “A Large-Scale Study of Failures in High Performance Computing Systems,” Proc. Int’l Conf. Dependable Systems and Networks (DSN).

J. Stearley and A. Oliner (2008), “Bad Words: Finding Faults in Spirit’s Syslogs,” Proc. Workshop Resiliency in High Performance Computing.

V. Tabatabaee and J. Hollingsworth (2007). Automatic Software Interference Detection in

Parallel Applications," Proc. Conf. Supercomputing (SC).

J.ichi Takeuchi and K. Yamanishi (2006). A unifying framework for detecting outliers and change points from time series. IEEE Transactions on Knowledge and Data Engineering.

L. Tang, C. Tang, L. Duan, Y. Jiang, C. Zeng, and J. Zhu (2008). Movstream: An efficient algorithm for monitoring clusters evolving in data streams. In Proceedings of Granular Computing.

L. Tang, C. Tang, Y. Jiang, C. Li, L. Duan, C. Zeng,and K. Xu (2008). Troadgrid: An efficient trajectory outlier detection algorithm with grid-based space division. In Proceedings of NDBC.

K. Vaidyanathan and K. Gross (2003), "MSET Performance Optimization for Detection of Software Aging," Proc. Int'l Symp. Software Reliability Eng. (ISSRE).

Jian Xu, Man-wu Xu (2009). A Performance Monitoring Tool for Predicting Degradation in Distributed Systems. The 2009 International Conference on Web Information Systems and Mining,11:669-673.

L. G. Valiant (1990). A bridging model for parallel computation. Communications of the ACM, 33(8):103-111.

R. Vilalta and S. Ma (2002). "Predicting Rare Events in Temporal Domains," Proc. Int'l Conf. Data Mining (ICDM).

K. Wu, K. Zhang, W. Fan, A. Edward, P. Yu (2014). RS-Forest: A Rapid Density Estimator for Streaming Anomaly Detection. Proc. Int'l Conf. Data Mining (ICDM).

L. Zheng, C. Zeng, L Li, Y. Jiang, W. Xue, J. Li, C. Shen, W. Zhou, H. Li, L. Tang, T. Li, B. Duan, M. Lei, P. Wang (2014). Applying data mining techniques to address critical process optimization needs in advanced manufacturing. In Proc. of ACM Special Interest Group on Knowledge Discovery in Data SIGKDD.