ADGS: Anomaly Detection and Localization based on Graph Similarity in Container-based Clouds

Chengzhi Lu[1,2], Kejiang Ye[1,*], Wenyan Chen[1], Cheng-Zhong Xu[3]

[1]*Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences*
[2]*University of Chinese Academy of Sciences*
[3]*Faculty of Science and Technology, University of Macau*
{*cz.lu, kj.ye, wy.chen*}*@siat.ac.cn, czxu@um.edu.mo*

***Abstract*—Docker container is experiencing rapid development with the support from the industry like Google and Alibaba and is being widely used in large scale production cloud environment. For example, Alibaba has deployed millions of containers for its internal business, and most of the online services are already migrated to the containers. Those services are usually very complex, spanning multiple containers with complex interaction and dependency relationship. Detecting potential anomalies in such a large container-based cloud platform is very challenging. Traditional detection models usually use system resource metrics like CPU and memory usage, but rarely consider the relationship among components, causing high false positive rate. In this paper, we present a novel Anomaly Detection and root cause localization method based on Graph Similarity (ADGS) in the container-based cloud environment. We first monitor the response time and resource usage of each component in the application to determine whether the system status is normal or not. Then, we propose a new mechanism to locate the root cause of the anomalies based on graph similarity, investigating the anomaly propagation rules among cluster components. We implement and evaluate our method in a container-based environment. The results show that the proposed method can detect and determine the root cause of anomalies efficiently and accurately.**

***Index Terms*—Cloud computing; anomaly detection; anomaly localization; graph similarity;**

I. Introduction

Cloud computing is a powerful platform to provide on-demand computing, storage and network resources to end users. Recently, container technology is experiencing rapid development due to the benefits such as more efficient, light-weighted, and flexible [1]. Many cloud service providers already adopt containers to provide cloud services. For example, Alibaba has deployed millions of containers for its internal business, and most of the online services are already migrated to the containers [2]. Compared with the traditional virtualization technology, containers can be deployed more quickly with less overhead.

On the one hand, the applications running in the cloud become more and more diverse and the application structure also becomes more complex. Multi-tier applications are very common in the cloud, spanning multiple containers with complex interaction and dependency relationship. On the other hand, multiple applications deployed on the same physical machine is common in the current container-based cloud environment [3]. Efficient performance management can effectively improve cluster resource utilization. However, the performance management of those multi-tier applications with different characteristics and unpredictable interactions is very difficult, especially for anomaly detection and localization.

Anomaly detection is to identify abnormal operational behaviors and system errors such as hardware failures, unusual use of resources, and so on. The traditional anomaly detection methods have some limitations. For example, many detection techniques only use system resource metrics such as CPU, memory and network I/O to detect anomalies, ignoring the application characteristics and structure [4]. However, the traditional anomaly detection methods based on system resource metrics fail to work in such environments due to the complex interaction of multi-tier applications running in containers. In addition, resource usage in the cloud is also dynamic and hard to predict, causing high false positive rate of the traditional methods. Therefore, for the container-based cloud platform, it is very important to design a scalable and accurate detection method taking the application structure into account.

In this paper, we propose a new Anomaly Detection method for container-based cloud platform based on Graph Similarity (ADGS). Based on the structure graph of the multi-tier application, we construct a system state graph for anomaly detection and location. The edge weight of the system state graph denotes the response time or system performance metrics. By calculating the similarity of the system state graphs at different moments, we implement the detection and location of application anomalies. The combination of response time and system performance metrics improves the accuracy of the location and detection of anomaly detection. We implement and evaluate our method in a container-based environment. Compared with the traditional solutions, the proposed method is more accurate and scalable. Experimental results show that the proposed method can detect numerous system anomalies and locate the root cause accurately.

The rest of the paper is organized as follows. We introduce the basic definition of response time and system state graph in

This work is supported by the National Key R&D Program of China (No. 2018YFB1004804), National Natural Science Foundation of China (No. 61702492), Equipment Pre-Research Foundation (No. 61400020403), Shenzhen Basic Research Program (No. JCYJ20170818153016513), and Alibaba Innovative Research (AIR) project.

*Corresponding author.

978-1-7281-2583-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICPADS47876.2019.00016

Section II. Section III presents our method. We evaluate our method and analyze the results in Section IV. In Section V, we introduce the related work about anomaly detection. Finally, we conclude the paper in Section VI.

II. Basic Definition

As discussed, it is difficult to accurately detect and locate anomalies in multi-tier applications running on the container-based cloud platform only by using system metrics. We propose the system state graph to reflect the dependency relationship between containers and physical machines. We combine the system state graph with the system metrics to detect and locate anomalies. And we also consider the component response time to reduce the detection instability caused by the fluctuation of the system metrics.

A. Component Response Time

For a multi-tier application, the component response time includes the component processing time and the backend response time.

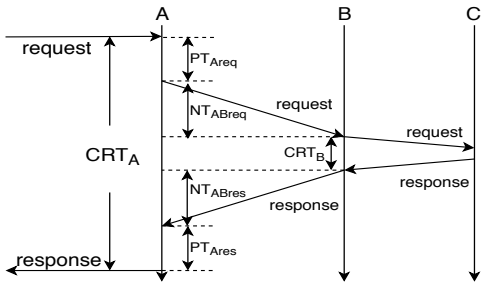


Fig. 1. Example of component response time.

Fig. 1 shows an example of how to calculate the component response time. In a simple three-tier application, the response time of component A is defined as follows:

$$CRT_A = CRT_B + NT_{AB_{req}} + NT_{AB_{res}} + PT_{A_{req}} + PT_{A_{res}} \quad (1)$$

CRT_A and CRT_B represent component response time of component A and B respectively. $NT_{AB_{req}}$ and $NT_{AB_{res}}$ represent the network transmission time of request and response between component A and B. $PT_{A_{req}}$ means the processing time of request from the outside of component A, and $PT_{A_{res}}$ means the processing time of response of component A. In this definition, we only focus on server-side processing time and network delay. Note that in this paper, the response time refers to the component response time.

Moreover, we call all components that depend on component A the *Downstream* components of A, whereas if they are dependent on A, they are referred to as *Upstream* components of A. In Fig. 1, A is a downstream component of B, and C is an upstream component of B.

B. System State Graph

In general, we can easily construct a graph through the deployment information of the dependency relationship among components and the relationship between components and physical machines. We call this graph *System State Graph*.

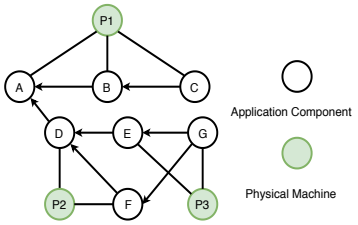


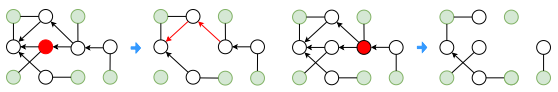
Fig. 2. Example of system state graph.

As shown in Fig. 2, we divide the vertexes of the graph into two types in order to locate the abnormal component, one is the vertexes representing the components which are running in containers, and the other is the vertexes representing the physical machines. Correspondingly, there are two types of edges of the graph. One is the *directed edges* representing the dependencies between components. The weight of the edge represents the component response time of the starting component. The other one is the *undirected edges* representing the affiliation of the components to the physical machine, and the weight of the undirected edge represents the resource utilization of the component.

We refer to the directed edge whose starting vertex is component B as the directed edge of B, and the undirected edge of B is the undirected edge that one of whose vertexes is B. In addition, there are some components in the application that are not upstream components of any other components. These components are called *output* components, such as Component A in Fig. 2. At the same time, there are also one or more components in the application that are not downstream components of any other components. These components are called *end* components, such as Component C and G in Fig. 2. In this paper, we assume that every service that the server provides requires the participation of the end component.

C. Anomaly Analysis based on System State Graph

According to the location and severity of the anomaly, the system state graph can have three different kinds of changes: *application component missing*, *physical machine missing* and *change of edge weight*. In this paper, we only consider the graph changes caused by the anomalies instead of manual operations such as shutting down the machine, changing the structure of application and so on.

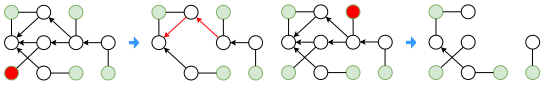


(a) With alternative components (b) Without alternative components

Fig. 3. Missing component vertex.

1) Application Component Missing: If the component encounters a serious failure, the component cannot respond to any request, and the vertex corresponding to the faulty component will disappear in the system state graph. In Fig. 3,

there are two different effects of vertex missing according to the location of component vertex: i) if there are multiple components with the same function on the same tier, the remaining components on the same tier will handle more requests, resulting in an increase in response time of remaining components; ii) if there are no other components with the same function in the same tier, all of the downstream components cannot get any response from the missing component, causing the service interruption.

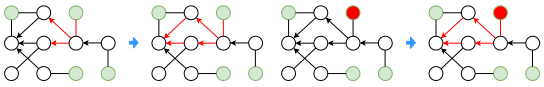


(a) With alternative components (b) Without alternative components

Fig. 4. Missing physical machine vertex.

2) Physical Machine Missing: If the physical machine encounters a serious fault, the physical machine may miss in the system state graph, all components running on this physical machine will be dropped. In the system state graph, the physical machine vertex and all component vertexes connected to this machine will disappear. If there are some alternative components, the weight of the directed edges of these alternative components will increase, such as the anomaly shown in Fig. 4(a). If these components have no alternative components, the downstream components of these components cannot get any response, like the anomaly in Fig. 4(b).

3) Change in Edge Weight: If a component encounters a slight anomaly, the component cannot use all resources allocated to it, which leads to a decrease in the running efficiency and an increase in the response time of this component. Eventually, the response time of all of the downstream components of this component will increase. This is represented in the system state graph as an increase in the weight of all the directed edges of the fault component and the downstream components of that component, as shown in Fig. 5(a). In addition, due to an anomaly of a physical machine, the response time of the components running on this physical machine increases, resulting in the increase in the response time of downstream components of the components running on this physical machine, as shown in Fig. 5(b).



(a) Weight changes of directed edges (b) Weight changes of undirected edges

Fig. 5. Weight changes of different types of edges.

III. Design of ADGS Method

In this section, we discuss the detailed design of Anomaly Detection and localization method based on system state Graph Similarity (ADGS). According to the above analysis, the anomaly can change the system state graph. Therefore, the anomaly detection and localization problem of the container-based cloud platform can be modeled as follows: Let $G_c = (V_c, E_c)$ denotes the system state graph of the current moment, and $G_p = (V_p, E_p)$ denotes the system state graph of the previous moment. The goal of the method is to calculate the similarity between G_p and G_c, and then return the abnormal vertexes.

A. Similarity of System State Graph

Generally, the measures of graph similarity are categorized into two types [5]: i) *Feature-based distance*. A set of features is constructed according to the description of the graph, and then these features are used to measure the similarity or distance of the graph. The feature-based distance include Euclidean distance, Frobenius distance [6] and so on; ii) *Cost-based distance*. The distance or similarity between two graphs means the steps that transforming one graph into the other. Graph edit distance [7] is a typical cost-based distance.

The change in the system state graph caused by the anomalies of components or physical machines are mainly divided into two categories: *change of structure* and *change of the edge weights*. i) *For structural change*, by comparing the differences between the vertex set and edge set in G_p and G_c respectively, we can easily get the system state and locate the root cause. If a vertex appears in G_p but disappears in G_c, we assume that the vertex has an anomaly. All the edges of that vertex disappear. This results in a significant difference in the structure of the system state graph between the two moments. ii) *For the change of edge weights*, according to the type of edge in the system state graph, we divide G_p into two subgraphs: one only contains directed edges and the other only contains undirected edges. They are denoted as G_{p_d}=(V_p,E_{p_d}), and $G_{p_{ud}}$=(V_p,$E_{p_{ud}}$), respectively. Similarly, G_c can also be divided into G_{c_d} and $G_{c_{ud}}$.

Firstly, we need to define the weight difference of the two edges. In the system state graph, the weight of the directed edges is a continuous value and the weight of the undirected edges has multiple dimensions. The weights of directed and undirected edges are in the same space, respectively. So we choose Euclidean distance to respectively measure the weight difference of edges [8]. The weight difference of the same edge at different moments is defined as follows:

$$Diff(e) = \frac{\|w_i(e) - w_j(e)\|}{max\{\|w_i(e)\|, \|w_j(e)\|\}} \tag{2}$$

where $w_i(u, v)$ denotes the weight of edge (u,v) in G_i and $w_j(u, v)$ means the weight of same edge in G_j.

Then, based on the weight difference of edges, we define the similarity between G_i and G_j as follows:

$$Sim(G_i, G_j) = 1 - \frac{\sum_{(u,v)\in G_i \bigcup G_j} Diff((u, v))}{|\ \{(u, v) \in G_i \bigcup G_j\}\ |} \tag{3}$$

B. Algorithm Design

The anomaly detection algorithm is depicted in Algorithm 1. Firstly, we determine whether there are vertexes missing in the

Algorithm 1 Anomaly detection and localization

Input: G_p=(G_{p_d},$G_{p_{ud}}$),G_c=(G_{c_d},$G_{c_{ud}}$)
Output: the state of system and the root cause of anomaly

1: **if** $|V_p|! = |V_c|$ **then**
2: **return** $v \in V_p - V_c$
3: **if** $\sigma \times Sim(G_{p_d}, G_{cd}) + (1-\sigma) \times Sim(G_{p_{ud}}, G_{cud}) < t$ **then**
4: $S_d = \{u, v | Diff((u, v)) > t_d, (u, v) \in G_{p_d}\}$
5: $S_{ud} = \{u^{'}, v^{'} | Diff(u^{'}, v') > t_{ud}, (u^{'}, v^{'}) \in G_{p_{ud}}\}$
6: **return** $v \in S_d \cap S_{ud}$
7: **return** normal

system state graph. If there are some vertexes missing, we then output the vertex difference of two graphs, which represents the root cause of the anomaly.

After that, using the similarity between the graph G_p and G_c, we can determine whether the service of the application in the current system is abnormal. And through the similarity between the graph $G_{p_{ud}}$ and $G_{c_{ud}}$, we determine whether the resource consumption of the application is abnormal. In the cloud platform, different applications have different sensitivity to resources and response time. So finally we sum up the similarity of the two subgraphs in proportion as the similarity between G_p and G_c. When the similarity between G_p and G_c is lower than a given threshold, the system is abnormal.

When the similarity of the system state graph is lower than the given threshold, the algorithm determines the location of the anomaly. The weight difference of each edge in the directed and the undirected subgraph are respectively sorted. The starting vertexes of the edges with the weight difference more than t_d in the directed subgraph are formed into the set S_d. We make the vertexes of the edges of the undirected subgraph with weight difference more than t_{ud} set to S_{ud}. The abnormal vertexes are the intersection of S_d and S_{ud}.

C. Time Complexity and Space Complexity

1) Time Complexity: The algorithm is mainly divided into two parts: one is to calculate whether the structure of system state graph changes; the other is to calculate whether the weight of the edges of system state graph change.

The structural similarity of two graphs is usually measured by the edit distance of the graph, which is NP-hard problem [9]. Note that the system state graph is constructed according to the dependencies of the components. All of the corresponding relationships of the vertexes and edges of two system state graphs depicting the same system can be easily acquired. Therefore different from the conventional structure similarity calculation of graphs, the similarity calculation of two system state graphs doesn't need to judge the corresponding relationship between vertexes and edges of the two graphs, which greatly reduces the complexity of system state graph similarity calculation. In Algorithm 1, the time complexity of calculating structural difference between the two system state graph is only related to the number of vertexes, which is $O(n)$ (n represents the number of vertexes). Similar to structural difference, we can calculate the weight differences between two system state graphs by traversing all edges one time, and its time complexity is $O(n^2)$. So the time complexity of Algorithm 1 is $O(n^2)$.

2) Space Complexity: The memory space for Algorithm 1 is mainly used to store the vertexes set, edges set, weights of edges and the weight differences of edges. Hence its space complexity is $O(n^2)$.

It is noted that we can calculate the graph similarity in a management machine. The agents collect the information of the components, such as response time, resource utilization and so on. Then agents send these information to the management machine. So the running of ADGS will not cause too much interference to the application.

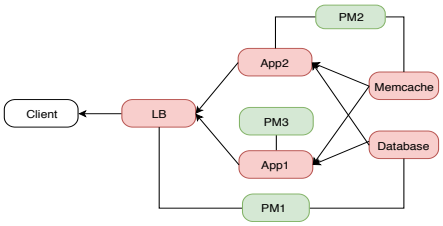


Fig. 6. Experimental setup.

IV. Evaluation

We evaluate our method with TPC-W benchmark [10], and deploy this benchmark using docker containers. We evaluate the system performance in two ways: 1) the anomaly detection ability; 2) the ability to locate anomaly. The detection and localization of physical machine missing and component missing anomaly are relatively simple, so we only analyze and evaluate the ability of the method to detect the anomaly of the weight changes of edges in this experimental section.

A. Experimental Setup

1) Environment: We deploy different components of the TPC-W benchmark in different containers in order to simulate a container-based application environment. Each container runs an agent to get the response time of the application components in the container, and each physical machine runs an agent to collect the resource utilization of containers. The experimental configuration is as follows:

- **Physical Machine**: Ubuntu 16.04, Docker 18.03-ce, 8v CPU, 16G RAM, 16G virtual memory
- **Container**: Ubuntu 16.04, 1v CPU, 2G RAM, 2G virtual memory
- **Application Benchmark**: TPC-W benchmark

As shown in Fig. 6, the whole benchmark application consists of five components in three levels, including one load balance server (LB server), two application servers (APP1 and APP2 server), one Memcached server (Mem server) and one database server (DB server). LB server and DB server run on physical machine 1, APP2 and Mem server run on physical machine 2, and APP1 server runs on physical machine 3.

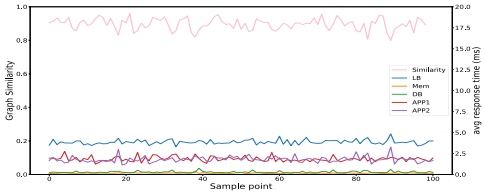


Fig. 7. Response time of each component and graph similarity in normal state.

2) Anomaly Types: We select the following types of common anomaly models to inject into the web application:

- **CPU Anomaly**: CPU is always occupied due to an anomaly in the system hardware or software.
- **Memory Anomaly**: Free memory is constantly being allocated and not released [5].
- **Network Anomaly**: There are two types of network anomalies. i) the decrease in available bandwidth caused by bandwidth being occupied, and ii) packets dropping caused by routing errors or link errors [11].

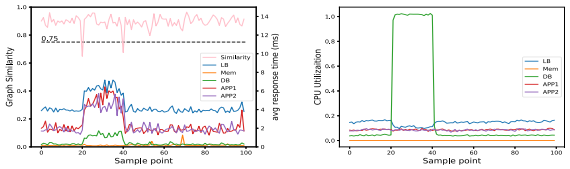
We simulate the CPU anomaly through a program that performs loop calculations [12] and simulate the memory anomaly through a program maintaining for several seconds that occupies lots of memory. As for network anomaly, only network anomaly caused by packets dropping due to routing errors or link errors is considered in this paper.

3) Sampling Interval: In this paper the sampling interval is set to 3 seconds to ensure real-time detection of anomalies and reduce the sampling overhead.

4) Value of σ: σ represents proportion of the directed subgraph in system state graph similarity. The larger the σ value is, the higher the proportion of the directed subgraph is. When σ is 0, the graph similarity reflects the normalized average value of the change in the utilization of cluster resources. When σ is closed to 1, it means that the similarity of directed subgraphs is mainly considered. Therefore, when the fluctuation of component response time is mainly considered, σ should take a large value; on the contrary, if the resource utilization is mainly considered, σ should be close to 0. In this paper, we set the value of σ to 0.5 so that the directed subgraph takes up the same proportion as an undirected subgraph.

B. Normal State

Firstly, we analyze the change of the graph similarity in the normal state of the application. In 300 seconds (about 100 sample points), the server processes requests from 50 clients concurrently. We take the average response time of component in each sampling interval as the response time. Fig. 7 shows the change in response time of different components under normal state and the change in the graph similarity. It can be seen that the graph similarities fluctuate with the response time, which is generally stable in the range of 0.8 to 0.9. We can also see that although the changes in response time are small, the graph similarity changes are relatively large.



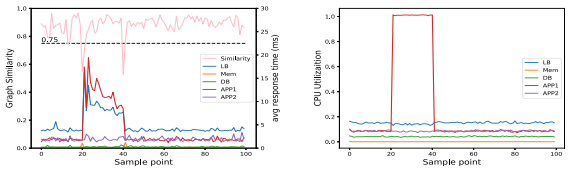
(a) (b)

Fig. 8. CPU anomaly in DB server: (a) Response time of each component and the system state graph similarity; (b) The CPU usage of each component.

C. Anomaly Detection Ability of ADGS

We compare the changes of component response time, resource utilization and graph similarity to show that the graph similarity has a high correlation with resource usage and response time and ADGS can detect anomaly well.

1) CPU Anomaly: Fig. 8 shows the changes in CPU usage, response time of each component and graph similarity after injecting the CPU anomaly into the DB server. The CPU usage of the DB server rapidly increases from 20% to 100% when the CPU anomaly is injected into the DB server and the response time of DB server increases immediately. Since DB server is the end component of the whole application, all downstream components of DB server have to wait for the response from DB server. The increase in the response time of DB server leads to an increase in the waiting time of downstream components. The increase in the waiting time causes the increase in response time of DB server. In addition, the increase in waiting time also leads to an increase in the CPU idle time of downstream components. So the CPU utilization of these components decreases. The changes in response time across the whole application and CPU utilization finally lead to great change in graph similarity, from 0.9 to 0.65, which exceeds the threshold.

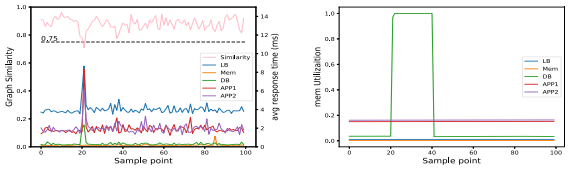


(a) (b)

Fig. 9. CPU anomaly in APP1 server: (a) Response time of each component and the system state graph similarity; (b) CPU usage of each component.

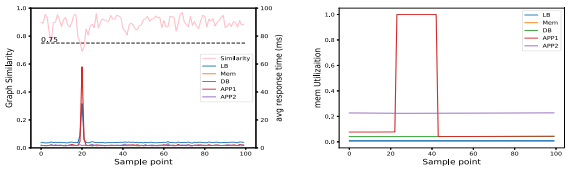
Fig. 9 shows the changes in CPU usage, response time of each component in the application and the fluctuation of the graph similarity after injecting CPU anomaly into APP2 server. Similar to the DB server, when the CPU anomaly occurs, the CPU usage of the APP1 server rises rapidly, and the response time of APP1 server also rises accordingly. And the impact of injecting CPU anomaly into APP1 server only propagates to the LB server. After the scheduling of the LB server, the impact of the APP1 server anomaly on the response time is gradually reduced. The CPU usage of other components is not decreased significantly. When the anomaly occurs, the graph similarity is about 0.55.

Therefore, when the CPU anomaly occurs, the graph similarity is much lower than that under the normal state, and



(a) (b)

Fig. 10. Memory anomaly in DB server. (a) Response time of each component and the change of system state graph similarity. (b) The memory usage of each component.



(a) (b)

Fig. 11. Memory anomaly in APP1 server: (a) Response time of each component and the change of system state graph similarity; (b) The memory usage of each component.

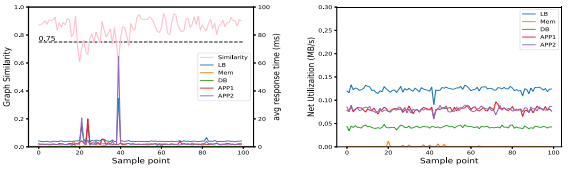
the ADGS can detect the occurrence of the anomaly by the change of the graph similarity.

2) Memory Anomaly: Fig. 10 shows the changes in response time, memory usage and the graph similarity of the application after injecting memory anomaly into the DB server. When the memory anomaly is injected into DB server, the memory usage of the DB server increases from below 10% to nearly 100%. At the moment when the anomaly is injected, the response time of all components increases significantly, and then the response time of components recover to be stable.

Compared with CPU anomaly, the memory anomaly of DB server does not significantly affect response time. In this paper, the effect of memory anomaly on response time occurs only at the moment of anomaly injection, and then the change in response time is slight. We speculate that when the anomaly occurs, the response time changes significantly because operating system takes up some of CPU time to handle memory page swapping. From the perspective of graph similarity, when the anomaly occurs, the graph similarity decreases greatly, and the lowest is around 0.7.

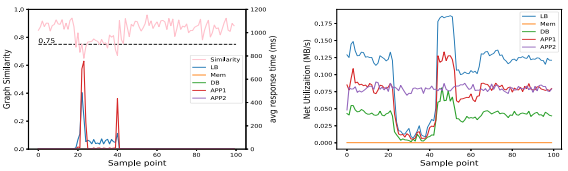
After we inject the memory anomaly into the APP1 server, the change of response time, memory usage and the similarity of system state graph is as shown in Fig. 11. Similar to the DB server, when memory anomaly occurs, the response time of the APP1 server changes greatly and leads to the response time of LB server on to increase. This causes the similarity of the system state graph to drop to around 0.7. The experiment shows that the ADGS can detect the memory anomaly well.

3) Network Anomaly: Fig. 12 shows the changes in response time, network inbound transmission rate of components in the application and graph similarity after injecting network anomaly into the DB server, respectively. When the network packets dropping anomaly is injected into the DB server, the network transmission speed of the DB server barely changes, and the network transmission speed of other components remained basically the same. But the response time changes a



(a) (b)

Fig. 12. Network anomaly in DB server: (a) Response time of each component and the change of system state graph similarity; (b) The network transmission speed (inbound) of each component.



(a) (b)

Fig. 13. Network anomaly in APP1 server: (a) Response time of each component and the change of system state graph similarity; (b) The network transmission speed (inbound) of each component.

lot. Observing the graph similarity, we can see that the value of graph similarity falls below 0.75 (nearly 0.6) in the period when the network anomaly occurs.

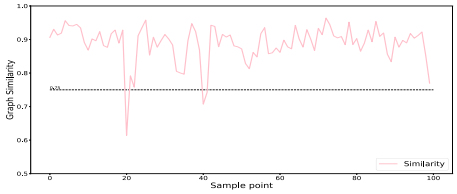
However, after injecting the anomaly into the APP1 server, the network utilization and response time of the application components are quite different from those of the DB server. As shown in Fig. 13, APP1 server's network utilization is significantly reduced when a network anomaly occurs on the APP1 server. As APP1 server traffic decreases, the network utilization of its upstream component DB server and downstream component LB server also decreases. However, the network traffic of APP2 server remains basically unchanged because of the normal state of LB, DB and Mem servers.

The response time of APP1 server and LB server increases significantly, while the response time of other components changes little when the anomaly occurs. The response time of APP1 server and LB server subsequently decline, but the response time of LB server remain high during the abnormal period while that of APP2 server is basically normal.

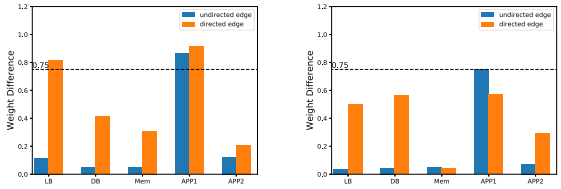
This can be clearly seen from the change in the graph similarity in Fig. 13(a). When the anomaly occurs, the graph similarity drops below 0.7, and the network anomaly can be detected accurately by the ADGS.

D. Anomaly Localization

When we inject a CPU anomaly into the APP1 server, as shown in Fig. 14(a), the graph similarity is reduced to below 0.75. At that moment, it can be seen that the difference values of the directed edge and the undirected edge of APP1 exceed the threshold, as shown in Fig. 14(b), and only the APP1 server meets this condition, so that the root cause can be determined to be the APP1 server. When the anomaly terminates, although the similarity of the graph is slightly less than the given threshold value, according to the difference value of each edge in the directed subgraph and the undirected subgraph respectively, there is no anomaly in any component.

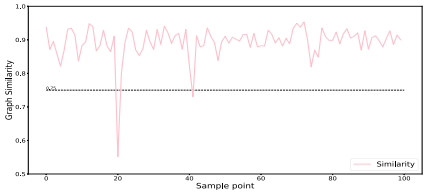


(a) Change of graph similarity

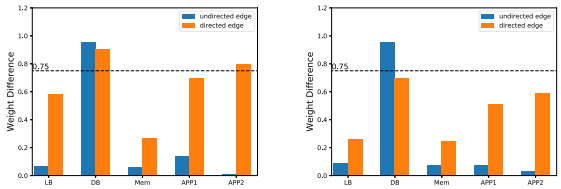


(b) Weight difference of edges at two abnormal points

Fig. 14. Inject CPU anomaly into APP1 server.



(a) Change of graph similarity

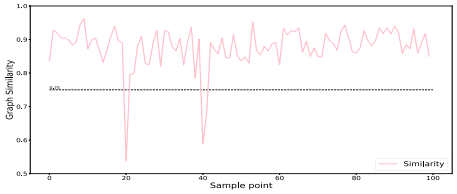


(b) Weight difference of edges at two abnormal points

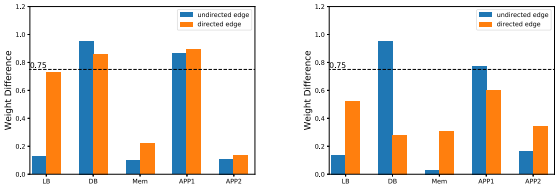
Fig. 15. Inject CPU anomaly into DB server.

When we inject a CPU anomaly into the DB server, there are two moments when the graph similarity is less than the threshold in Fig. 15(a). As can be seen from Fig. 15(b), the ADGS can determine the component where the anomaly occurs. From the perspective of response time, since the DB server is the end component, except the Mem server, the response time of all other components increases with the increase in the response time of DB server, and the specific localization where the anomaly occurs cannot be effectively identified. However, by combining the undirected graph and the directed graph, we are able to accurately locate the abnormal position.

Finally, we inject CPU anomaly into both the APP1 server and the DB server. It can be seen that when the CPU anomaly occurs, the similarity of the system state graph of the application is reduced to below 0.55. By analyzing the directed and undirected edges in the system state graph, the ADGS considers LB server as a normal component because the undirected edge is less than 0.2 although the difference



(a) Change of graph similarity



(b) Weight difference of edges at two abnormal points

Fig. 16. Inject CPU anomaly into DB server and APP2 server.

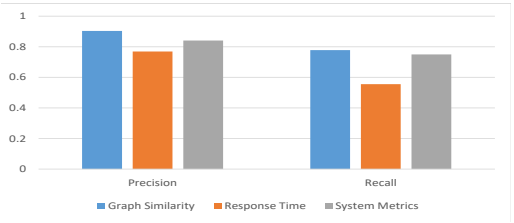


Fig. 17. Precision and recall.

value of the directed edge of LB server is close to 0.75. DB server and APP1 server are determined as an abnormal component while the difference value of the directed and undirected edge of them are both exceed the given threshold. Thus ADGS can locate multiple anomalies simultaneously.

E. Precision and Recall

Precision and recall are important evaluation indicators of the algorithm. The precision of algorithm in this paper only considers whether the algorithm can detect the anomaly when it occurs and whether the detected anomaly is correct. We will randomly inject anomaly into the system to evaluate the precision of anomaly detection and localization of the algorithm. At the same time, we compare the methods based on response time and system metrics, respectively. As shown in the Fig. 17, we can see that the ADGS has high precision and recall in detecting and locating anomaly. The precision is about 0.9 and the recall is closed to 0.8. Compared with the anomaly detection and localization method based on response time, the precision and recall rate of the ADGS are higher. So the ADGS can detect and locate the anomaly more effectively compared to the traditional anomaly detection methods.

V. Related Work

There is much existing work about detecting anomalies in the cloud platform. The current cloud platform anomaly detection methods are divided into two types: anomaly detection methods based on i) system metrics and ii) response time.

Most anomaly detection methods based on system performance metrics use system performance monitoring tools to collect system resource utilization such as CPU and memory utilization, disk I/O rate, network I/O rate [13]–[16], etc, and then the anomaly is detected by comparison to a given threshold. Some methods use statistical tools [17] or machine learning methods, such as Bayesian networks [18], entropy [19], clustering [20], Markov Chain [21], to analyze system metrics for determining whether the system is abnormal.

The anomaly detection methods based on response time are also widely studied due to its light weight and scalability. Certes [22] analyzes the network delay by analysing the control message of the TCP protocol to determine the abnormal state of the system. PBAD [23] determines whether the service is abnormal by comparing the measured response time of the service with the estimated response time.

There is also some work paying attention to the graph comparison and graph similarity in different domain. Graph isomorphism [24] discusses the problem of whether two graphs are isomorphic to each other, or whether one is isomorphic to subgraph of the other. However, due to the problem is NP-Complete (graph isomorphic problem) or provably NP-Complete (subgraph isomorphic problem), the algorithm of graph isomorphism takes a lot of time, so it could not be used in real scenarios well. Papadimitriou etc. [25] research some methods to detect anomaly in web graph. They compare several common rules on the Web graph, and optimize the algorithms that applied these rules. But the web graph is only built at the application layer, lacking consideration for system resource usage, and can't be applied to the container cloud.

There are also many studies on the anomaly localization of cloud platforms. FChain [26] finds the mutation points by using Fast Fourier Transform on the state of resource utilization, and simultaneously locates the anomaly in combination with the propagation of the anomaly. CloudPD [27] clusters system performance metrics by k-nearest neighbor algorithm to find anomalies and locate anomalies.

VI. Conclusion

We proposed an anomaly detection and localization method based on graph similarity. The traditional detection methods based on system resource utilization or response time only consider some of the anomaly factors, and cannot reflect the overall state of the application. So their accuracy is not high. Our method combines system resource utilization, dependency relationship of components in the multi-tier application and response time. We implement and evaluate our method by using the multi-tier application benchmark. Experiments show that our method can detect multi-tier application anomalies accurately. It also shows that our method can accurately locate anomalies using graph similarity. The precision of our method is up to 0.9 and the recall of the method is closed to 0.8.

References

[1] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: a state-of-the-art review," *IEEE Transactions on Cloud Computing*, 2017.

[2] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on alibaba cluster trace," *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2884–2892, 2017.

[3] W. Chen, K. Ye, Y. Wang, G. Xu, and C.-Z. Xu, "How does the workload look like in production cloud? analysis and clustering of workloads on alibaba cluster trace," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 102–109.

[4] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, pp. 2093–2115, 2013.

[5] L. Abeni, A. Goel, C. Krasic, and J. Snow, "A measurement-based analysis of the real-time performance of linux," in *Eighth IEEE Real-time & Embedded Technology & Applications Symposium*, 2002.

[6] M. Grohe, G. Rattan, and G. J. Woeginger, "Graph similarity and approximate isomorphism," *CoRR*, vol. abs/1802.08509, 2018.

[7] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Analysis and Applications*, vol. 13, pp. 113–129, 2008.

[8] H. Bunke, P. J. Dickinson, M. Kraetzl, and W. Wallis, *A Graph-Theoretic Approach to Enterprise Network Dynamics*, 01 2007, vol. 24.

[9] S. Bougleux, L. Brun, V. Carletti, P. Foggia, B. Gaüzère, and M. Vento, "Graph edit distance as a quadratic assignment problem," *Pattern Recognition Letters*, vol. 87, pp. 38–46, 2017.

[10] tpcw, "[online]," http://www.tpc.org/tpcw/, accessed December 4, 2018.

[11] M. Roesch, "Snort - lightweight intrusion detection for networks," *Proc Usenix System Administration Conf*, pp. 229–238, 1999.

[12] N. J. Gunther, *Analyzing Computer System Performance with Perl::PDQ*, 2011.

[13] T. Wang, J. Xu, W. Zhang, Z. Gu, and H. Zhong, "Self-adaptive cloud monitoring with online anomaly detection," *Future Generation Computer Systems*, vol. 80, pp. 89–101, 2018.

[14] K. Ye, Y. Liu, G. Xu, and C.-Z. Xu, "Fault injection and detection for artificial intelligence applications in container-based clouds," in *International Conference on Cloud Computing*. Springer, 2018, pp. 112–127.

[15] S. Ji, K. Ye, and C.-Z. Xu, "Cmonitor: A monitoring and alarming platform for container-based clouds," in *International Conference on Cloud Computing*. Springer, 2019, pp. 324–339.

[16] R. Ravichandiran, H. Bannazadeh, and A. Leon-Garcia, "Anomaly detection using resource behaviour analysis for autoscaling systems," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, June 2018, pp. 192–196.

[17] J. Hochenbaum, O. S. Vallis, and A. Kejariwal, "Automatic anomaly detection in the cloud via statistical learning," 2017.

[18] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *IEEE International Conference on Distributed Computing Systems*, 2012.

[19] C. Wang, K. Viswanathan, C. Lakshminarayan, V. Talwar, W. Satterfield, and K. Schwan, "Statistical techniques for online anomaly detection in data centers." in *Ifip/ieee International Symposium on Integrated Network Management*, 2011.

[20] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, pp. 15:1–15:58, 2009.

[21] W. Sha, Y. Zhu, M. Chen, and T. Huang, "Statistical learning for anomaly detection in cloud server systems: A multi-order markov chain framework," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2018.

[22] D. P. Olshefski, J. Nieh, and D. Agrawal, "Inferring client response time at the web server," *Acm Sigmetrics Performance Evaluation Review*, vol. 30, no. 1, pp. 160–171, 2002.

[23] J. Kim and H. S. Kim, "Pbad: Perception-based anomaly detection system for cloud datacenters," in *IEEE International Conference on Cloud Computing*, 2015.

[24] P.-A. Champin and C. Solnon, "Measuring the similarity of labeled graphs," in *ICCBR*, 2003.

[25] P. Papadimitriou, A. Dasdan, and H. Garcia-Molina, "Web graph similarity for anomaly detection," *Journal of Internet Services and Applications*, vol. 1, pp. 19–30, 2010.

[26] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "Fchain: Toward black-box online fault localization for cloud systems," 2013.

[27] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "Cloudpd: Problem determination and diagnosis in shared dynamic clouds," in *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2013.