Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/070653

Online Anomaly Detection for Service-Oriented Components in OSGi-based Applications

Tao Wang[1,2,3,*], *Jun Wei* [1,2], *Wenbo Zhang*[2] *and Hua Zhong*[2]

[1] State Key Laboratory of Computer Science, Beijing 100190, P.R. China
[2] Institute of Software, Chinese Academy of Sciences, Beijing 100190, P.R. China
[3] University of Chinese Academy of Sciences, Beijing 100049, P.R. China

Received: 26 Feb. 2013, Revised: 24 Jun. 2013, Accepted: 25 Jun. 2013
Published online: 1 Nov. 2013

Abstract: OSGi has become one of the most promising frameworks for managing service-oriented and component-based applications. The OSGi-based service-oriented components delivered by different vendors are usually black-box program units which lack source code and design documents. Thus, it is difficult to evaluate their quality by static code analysis, and the defective components may lead to the failure of the whole system eventually. In this paper, we propose an online method for detecting anomalous service-oriented components in OSGi-based applications. A thread-tracing method is presented to monitor resource utilization and interactions between components. The method considers both the dynamic service invocation and static method invocation. Furthermore, according to the monitored data, we detect anomalous components by control charts, which can detect the anomalous trend of resource utilization without prior knowledge. A prototype tool was implemented and applied to a real application server. The experimental results show that our method 1) is of high accuracy for monitoring resource utilization in component perspective; 2) does not introduce significant overhead; 3) and can detect anomalous components effectively.

Keywords: Anomaly detection, service-oriented component, OSGi, resource utilization, control charts.

1 Introduction

The component-based software engineering greatly improves the efficiency and quality of software development; organizations always adopt it for developing large-scale complex software [1]. In recent year, OSGi (Open Service Gateway initiative) has become one of the most promising frameworks for managing service-oriented and component-based applications [2]. The OSGi framework, which provides a service model and a service registry, is an execution environment for dynamically loadable services. OSGi technology is attracting growing interest, and a large number of large-scale projects have released new versions with OSGi, such as JEE application server Websphere, IDE eclipse and the BMW automobile control system. The services based on OSGi are always implemented as bundles that are service-oriented components [3]. The COTS (Commercial Off-The Shelf) market around OSGi is emerging, where the number of third party components is increasing [4]. However, a defective component may affect all the related components and lead to the failure of the whole system eventually. Thus, it is a critical issue for COTS to ensure the quality of components [5].

However, since the COTS components are usually black-box program units which lack source code and design documents, it is difficult to understand the characteristics of components, and evaluate their quality by static code analysis. Furthermore, some runtime factor, e.g., access sequences, concurrency number and resource usage, may cause contextual anomalies [6], which are difficult to be eliminated through testing. Therefore, detecting anomalous components online is essential for improving the reliability of OSGi-based applications.

This paper proposes an online method for detecting anomalous service-oriented components in OSGi-based applications. The main contributions of this paper are as following:

–A thread-tracing method is proposed to monitor resource utilization and interactions of components. It is an online method, which neither modifies software nor introduces significant overhead.

* Corresponding author e-mail: wangtao08@otcaix.iscas.ac.cn

–The control charts for resource utilization are introduced to detect anomalous components. They can detect the anomalous trends of resource utilization without prior knowledge.
–An anomaly detection framework for OSGi-based applications is presented. A prototype tool is implemented and applied to a real application server.
–The experimental results demonstrate that our method can monitor resource utilization in high accuracy without significant overhead, and detect the anomalous components effectively.

The rest of this paper is organized as follows. Section 2 presents a thread-tracing based method for monitoring components. Section 3 introduces control charts to detect anomalous components. Section 4 demonstrates the design and implementation of the prototype tool. Section 5 provides experimental results to validate the method in accuracy, overhead and effectiveness. Section 6 presents our discussion and future work. Section 7 reviews the related works, followed by conclusion in Section 8.

2 Monitoring components in OSGi-based applications

An OSGi service platform is composed of service providers, service requesters and a service registry. A service provider registers services to publish, and a service requester discovers services from the service registry to invoke. The service described as a Java interface is always packaged as a standard JAR file, namely bundle, in which service implementation, related resource files and manifest files are included. Bundles interact with each other as service invocation. Since bundles are basic management units in OSGi, we take them as monitored targets.

We analyze component-based applications from two perspectives that are performance metrics of a single component and interaction behavior between components [7]. Since CPU and memory utilization are important properties for evaluating a software component [8], we present a method to monitor these performance metrics.

2.1 Monitoring CPU utilization of a bundle

A thread is the basic unit to which the operating system allocates processor time. Thus, the CPU utilization of a bundle is the sum of the CPU time consumed by different threads, which execute within the same bundle. We have two monitoring perspectives as follows.

–Bundle perspective. Threads are grouped into different bundles, and each thread belongs to a specific bundle. Thus we add the CPU time of every thread in the bundle.
–Thread perspective. A thread execution is divided into some stages, each of which belongs to a specific bundle. Thus we add the CPU time of every stage in the bundle.

Because of frequent interactions between bundles by invocations, the relations between bundles and threads vary dynamically. If we follow the first perspective, the thread schedule model should be modified, and significant overhead will be introduced as presented in [9]. Therefore, we adopt the second choice through tracing thread transfer between bundles.

It is easy to calculate CPU time utilized by a thread during a period using the JVMTI provided by the JVM (Java Virtual Machine). Thus, how to divide the CPU time of a thread into different bundles becomes an essential question to answer. As is shown in algorithm 1, we describe the method for monitoring bundle CPU utilization.

Step 1. Initialize bundle ID for every thread.

Step 1.1. When a bundle is initialized, OSGi invokes start () method in the Activator class to start the bundle. We set the bundle ID of the thread as the started bundle through labeling the thread before and after the start method in the OSGi platform.

Step 1.2. When a new thread is created, we set the bundle ID of the thread as that of its parent thread.

Step 2. Trace thread transfer between bundles.

Decide whether thread transfer happens. When a service is invoked, if the service provider and the service consumer are in different bundles, thread transfer happens.

Step 3. Calculate CPU time of bundles.

If the bundle ID of a thread varies after entering the invoked service, the CPU time is calculated and added to the original bundle, and the time stamp is updated. After exiting from the service, the CPU time is calculated and added to the invoked bundle.

2.1.1 Monitoring dynamic service invocation

There are two kinds of component interactions that are dynamic service invocations and static method invocations. As for the service invocation, we use an event-driven mechanism to trace service invocations through listening to the events in the service registry, as is shown in the Figure 1. To avoid affecting the execution code in the original bundle and deal with the arriving services during execution, we create a proxy object for every required service. A proxy class is generated when a service is registered, and the proxy class is instantiated when the service is invoked. We also modify the service registry to redirect service requests to the service proxy. Thus, the proxy object instead of the original service provides service for a service consumer transparently. In every proxy class, the monitoring point is inserted before and after the service invocation to label the changed bundle ID of a thread.

Algorithm 1 Monitoring CPU utilization

Input:
OSGi-based applications;
Output:
CPU utilization for every bundle;
1: **if** Listen to the event(bundle b is initialized) **then**
2: Set the bundle ID of thread t as b, and the time stamp for t ;
3: **end if**
4: **if** Listen to the event(thread ts is initialized) **then**
5: Get the current thread tp which is the parent of ts, and the bundle ID of tp as bp;
6: Create a new thread ts, and Set the bundleID of ts as bp;
7: **end if**
8: **if** Listen to the event(service s_i invoke service s_j) **then**
9: Get the bundle ID of s_i as b_i , and the bundle ID of s_j as b_j ;
10: **if** $b_i != b_j$ **then**
11: Get $bundleInfo_i$ of b_i, and current thread t;
12: Calculate $bundle\_Info_i.CPUTime+ = t.Calculate\_CPU()$;
13: Set time stamp for t, and bundle ID of thread t as b_j;
14: Execute s_j;
15: Get $bundle\_Info_j$ of b_j;
16: Calculate $bundle\_Info_j.CPUTime+ = t.Calculate\_CPU()$;
17: Set time stamp for t, and bundle ID of t as b_i;
18: **end if**
19: **end if**

2.1.2 Monitoring static method invocation

Although the OSGi specification recommends developers to implement the interactions between components with service invocation, some developers used to invoke the functions from other components with the traditional static method invocation. Thus we propose an AOP (Aspect-Oriented Programming) based method [10] to trace the thread transfer between components. The OSGi framework analyzes the meta data file recording the exported packages automatically, when a bundle is loaded dynamically. We extend the original OSGi framework, so that it reports the exported methods to our monitoring tool when the analysis is finished. Our tool decides whether the classes being loaded are exported by the bundle according to the report. Then, we use AOP to insert the monitoring points into the beginning and the end of the public method in the class which exports methods.

We note that some exported packages are not invoked by other components. The thread transfer does not happen, when the invoking method and the invoked method in the same bundle. However, the redundant monitoring points introduce unnecessary overhead. Therefore, we use a static code analysis method to reduce the number of monitoring points before weaving class. Method invocations usually take the form of "targetObj.methodName(parameters)"; the key to analyzing the calling relationship is to know the possible types of the objects which the targetObj may point to. We use the class hierarchy analysis method [11] to gain this knowledge. All the subtypes of targetObj's type are among the possible types, and we can get all the possible target methods denoted by the methodName. With the knowledge of the target methods in every invocation statement, we can easily acquire the calling relationship between methods, classes, and packages. If two packages in two different bundles have the calling relationships in OSGi, the corresponding imported and exported packages should be specified in the metadata files of the two bundles. Thus we do not weave the methods invoked in the same bundle to reduce monitoring overhead.

The object of our AOP based method is to trace the thread transfer between components, when a component invokes the others with the static method invocation.

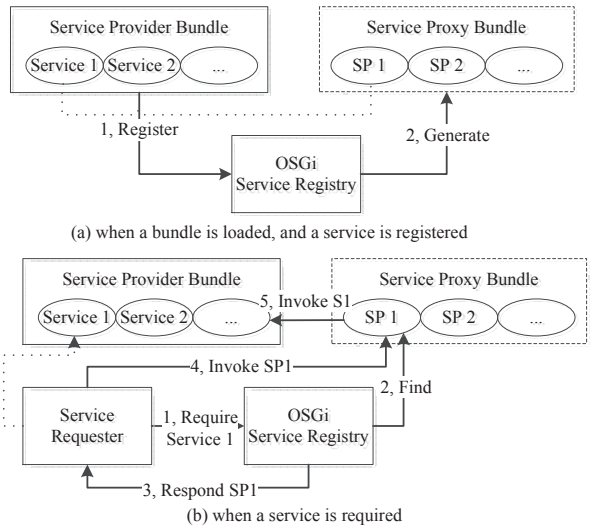

Fig. 1: Service proxy generation

2.2 Monitoring memory utilization of a bundle

The system memory is occupied by objects, which we will categorize into different bundles. We can also use the JVMTI to tag every object and calculate its size in memory. The objects created by different bundles are tagged with bundle IDs, and we calculate the memory utilization for every bundle. However, how to distinguish which bundle the objects are subordinate to is a key problem. With the algorithm 1, we get the relations between bundles and threads in different periods, and objects are created by threads. Thus we take the thread as a bridge between objects and bundles to locate the objects belonging to different bundles.

In essence, the CPU monitoring and memory monitoring are both to locate the units in their bundles. The differences are as follows: 1) the memory utilization

is gotten when collecting the calculation result, but the CPU utilization is calculated incrementally throughout the whole monitoring process; 2) an object always belongs to the bundle which created it until the object is destroyed, while thread frequently transfers between bundles.

2.3 Monitoring interactions between bundles

Since OSGi is a service-oriented platform, we focus on the service interactions between bundles. We use the bundle ID of a thread before and after a service invocation to trace bundle interactions. An event-driven mechanism is designed by listening to events in the service registry to update the service interaction graph at runtime.

Algorithm 2 describes the method for service interaction graph generation. We listen to the events in the service registry. When the service provider registers the service in the registry, we create one node in the graph to represent a bundle. At the same time, when the service consumer requests a service, a directed edge is created to connect the service provider and service consumer bundles. Next, we record the behavior of bundles through increasing the weight on the edge. During the execution period, as introduced in CPU monitoring, we can find the bundle IDs of a thread before and after a service invocation. If the service provider bundle tagged as A and a service consumer bundle tagged as B are not in the same bundle, we increase weight on the edge connecting from A to B.

Algorithm 2 Monitoring service interactions

Input:
service s_i;
service s_j;
Output:
service interaction graph g;
if Listen to the event(s_i invokes s_j) **then**
  $s_{i,j}++$;
end if
if $node_i$ == NULL **then**
  Create $node_i$ in g ;
end if
if $e_{i,j}$ == NULL **then**
  Create $e_{i,j}$ in g ;
  Set the value of $e_{i,j}$ as 1;
else
  Get current thread t, and bundle ID b_i of t;
  Enter service s_j ;
  Get bundle ID b_j of s_j;
  if b_i!=b_j **then**
    $e_{i,j}++$;
  end if
end if

3 Anomaly detection with control charts

According to the monitored resource utilization and interactions of components, we can further detect anomalous components. The metrics of resource utilization help developers to evaluate components, and find underlying problems, for example CPU exhaustion caused by an endless loop, or out of memory incurred by frequently allocating objects without freeing immediately. A control chart is a statistical tool used to distinguish between variation in a process resulting from common causes and variation result-ing from special causes [12]. Thus we use control charts to detect the symptoms of re-source utilization. Our goal of using control charts is to detect whether the resource utilization of components is stable or not. The stability is defined as a state in which the resource utilization has displayed a certain degree of consistency in the past, and is expected to do so in the future.

Control charts monitor component resource utilization, and raise an alarm if the metrics are not in stability. For example, in an application server, the memory utilization of a web container should be kept within a reasonable range under stable workload. When a problem happens, e.g., memory leak, the memory utilization of the web container shows anomalous trends. Then control charts will detect the gradually increasing trend in memory utilization of the component, even if it is still within a reasonable range. We make use of the XmR (Individual X and Moving Range) control charts, in which the individual (X) chart displays individual measurement, and the moving range (MR) chart shows variability between one data point and the next. Two XmR charts are employed to represent CPU and memory utilization respectively for every component to detect anomalies as follows:

Step 1. Collect CPU and memory utilization of every component in period. Since the resource utilization is usually in proportion to the number of service invocations, we make use of m_i as an individual measurement to develop control chart, where m_i is the CPU/memory utilization of a component, and k is the number of invocations during a period.

Step 2. Calculate the overall average of the individual measurements.Let's denote:

$$\bar{x} = (x_1 + x_2 + \cdots + x_n)/n, \tag{1}$$

, Where $\bar{x}$ is the average of the individual measurements, x_i is an individual measurement, and n is the number of measurements.

Step 3. Calculate the average of the moving ranges. The average of all moving ranges becomes the centerline for mR-chart.Let's denote:

$$\bar{mR} = (mR_1 + mR_2 + \cdots + mR_n)/n, \tag{2}$$

$$mR_i = |x_{i+1} - x_i|, \tag{3}$$

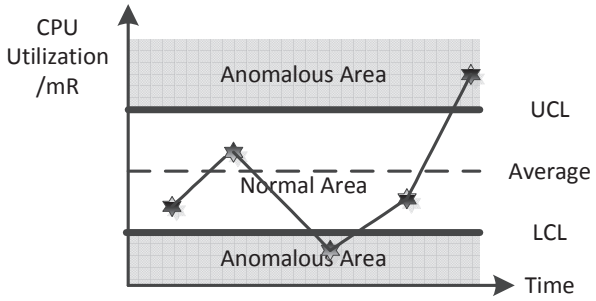

Fig. 2: Example of control chart

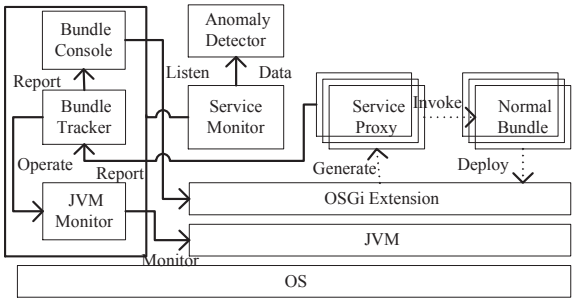

Fig. 3: Bundle monitoring tool architecture

where $\bar{mR}$ is the average of the individual moving ranges, mR_i is an individual moving range measurement, and n is the number of measurements.

Step 4. Calculate the Upper and Lower Control Limits (UCL/LCL) for the individual measurements to get the X-chart. To find these control limits, we use the following formula:

$$UCL_x = |\bar{x} + \alpha \bar{mR}|, \tag{4}$$

$$LCL_x = |\bar{x} - \alpha \bar{mR}|. \tag{5}$$

Step 5. Calculate the upper and lower control limits for the moving ranges to get the mR-chart. To find these control limits, we use the following formula:

$$UCL_{mR} = \beta \bar{mR}, \tag{6}$$

$$LCL_{mR} = None. \tag{7}$$

According to the statistics theory, we use constant α and β which are specified as 2.66 and 3.268 respectively. Figure 2 gives an example of anomaly detection with the control charts. The x-axis represents a sampling period, and the y-axis represents CPU utilization and mR. The scale between the LCL and UCL is regarded as the normal area, while the other scales are regarded as the anomalous areas. The system is detected in an anomaly status when the monitored points occur in the anomalous area. For example, the first, second and fourth points are in the normal area, while the third and fifth points are in the anomalous area. Thus we can get XmR control charts, and any point out of the normal scale, which is between the LCL and UCL, will be detected as an anomaly.

4 Anomaly detection system implementation

The system architecture of our prototype is illustrated in Figure 3. It is composed of three parts that are anomaly detector, OSGi extension, and service monitor including JVM monitor, bundle tracker and bundle console. The JVM monitor is responsible for labeling threads and objects with bundle ID, and calculating their resource utilization. The bundle tracker is a bridge between Java applications implemented with Java on top of OSGi and the JVM monitor implemented with native code. The OSGi extension extended from standard OSGi generates proxies for registered services, and redirects service requests to proxies. The anomaly detector analyzes the monitored data collected from the service monitor.

4.1 OSGi extension

OSGi extension is the extension of OSGi kernel, which is responsible for generating proxy objects. The ASM which is an operation framework of Java bytecode is used to generate new class or enhance an existing class. In the service registry, we use ASM to generate the service proxy class which provides service interfaces to invoke service objects. When the service consumer requires the services from the service provider, the service consumer gets a service proxy object initiated by the service proxy class. When the service is invoked, the corresponding service in a proxy object is invoked. In the beginning and ending of the service invocation, the thread information is updated to calculate CPU time.

In order to reduce system resource overhead and improve performance, our system provides an interface for users to customize monitored bundles. If the service is not in the monitored service list, the proxy object is not generated, and the original service object provides service as usual. The service proxy object is a new service introduced by our monitoring method, which is generated by the OSGi extension using the bytecode injection to trace the thread transfer between bundles. Felix is an Apache open source project, which implements the OSGi R4 kernel specification. Since Felix is stable and simple, it has been adopted by large scale projects such as JonAS , GlassFish , NetBean , thus we adopt it as the OSGi kernel to implement the OSGi Extension.

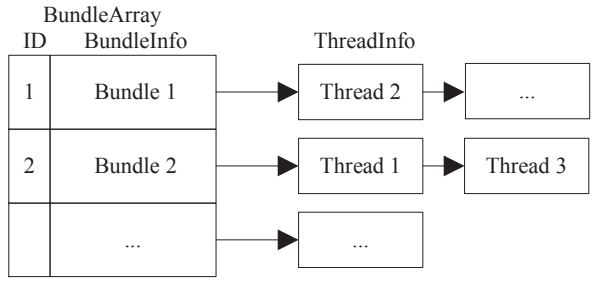

Fig. 4: JVM monitor data structure

4.2 JVM monitor

The responsibility of the JVM monitor is listed as follows.

–Tagging Thread: a thread is tagged as bundle ID, when the thread is created or transferred.
–Tagging Object: an object is tagged as the bundle ID of the bundle which creates it.
–Calculating Resource Utilization: CPU time and memory utilization of the bundle are calculated.

The JVM monitor monitors JVM through JVMTI, especially traces threads. Java applications take the bundle tracker as a bridge to communicate with JVM monitor, so the JVM monitor is transparent and does not conflict with the Java applications. The JVM monitor exists as a dynamic-link library, which is loaded when an application starts.

The monitoring data is stored in an array as BundleArray used to store bundle information, and we take bundle ID as the index of the bundle information with BundleInfo object. The information on threads as ThreadInfo object is saved in the BundleInfo, and organized as a chain, as is shown in Figure 4. The JVM monitor subscribes to target events in the JVM, and the method is invoked by callback function when the events happen as follows.

–START-THREAD: a ThreadInfo object is created, tagged with BundleID and linked in the chain of ThreadInfo in BundleInfo.
–END-THREAD: a ThreadInfo object is found,the thread CPU time is added to the BundleInfo, and the ThreadInfo object is deleted.
–CREATE-OBJECT: the object is tagged.

4.3 Bundle tracker and bundle console

Since tagging objects and threads requires entering the native code from Java code, the bundle tracker builds a bridge between Java applications and native code. Java applications are able to communicate with native code by invoking some method in the bundle tracker. We adopt JNI which is a standard Java API to integrate Java with other program languages in the bundle tracker. The bundle tracker module is composed of two bean classes that are BundleCPUInfo and BundleMemInfo to record Bundle CPU and memory information. When a Java application sends requests to JVM monitor, an ArrayList of bundle information is replied. Bundle console which is a Java application bundle on the JVM provides graphic interfaces for users to show the resource utilization and service interaction for every monitored bundle. Users are able to observe the status of every bundle, and do some operations such as install, uninstall, start, stop, or update the bundles running on the OSGi platform conveniently.

The service interaction graph implemented in the bundle console is a dynamic graph, so we adopt an event-driven mechanism by listening to events in registry to update the service interaction graph at runtime.

The events and their operations are listed as follows.

–REGISTER-SERVICE: if a node standing for the bundle, which the service is attached to, does not exist, the node and the service are created.
–UNREGISTER-SERVICE: the service and its edges are deleted. If the node only has the service, we also delete the node and the connected edges. Otherwise, the weight on the connected edges is decreased.
–GET-SERVICE: if the edge from the invoking node and the invoked node exists, the weight on the edge is increased. Otherwise, the edge is created.
–UNGET-SERVICE: the weight on the edge from invoking node and invoked node is decreased.

5 Evaluation

5.1 Monitoring accuracy

Since there is no standard benchmark for evaluating the accuracy of our CPU/memory monitoring method, we implemented a simulation to get a better understanding of the accuracy. As is shown in Figure 5, the simulation consists of eleven bundles including a controller bundle and ten service bundles. Each of these service bundles implements a service whose function is to loop for a fixed period, and the controller bundle invokes these services and counts the time spent on each service. These bundles are deployed on the OSGi platform, in which their services are registered. The controller bundle invokes the service bundles for several loops in random order. The service is assigned a quantitative execution time respectively, from 5 milliseconds till 50 milliseconds, as is described in the x-axis of Figure 6. The controller bundle invokes services for 10, 30, 50, 70, 90 and 110 loops in six experiments. Obviously, the expected CPU time is the product of service time and the number of loops in each experiment, and then we compare the expected CPU time with the monitored measurement.

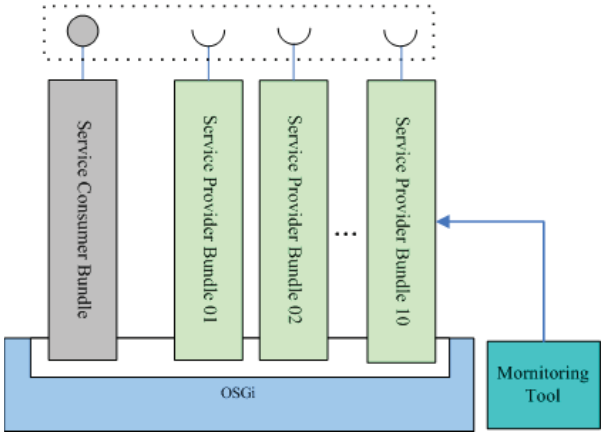

Fig. 5: Accuracy evaluation environment

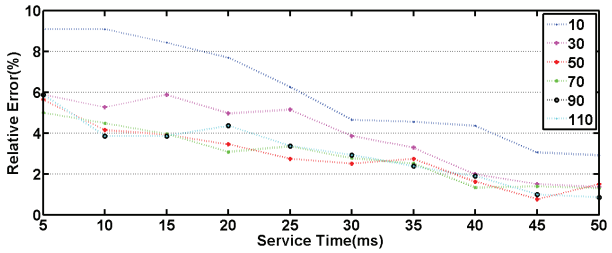

Fig. 6: CPU monitoring accuracy

The accuracy of monitoring CPU time in terms of relative error is shown in Figure 6, in which the curves indicate the results of different experiments in which services loop for 10, 30, 50, 70, 90 and 110 times. It is seen that the relative error decreases with the service time increases. This is explained that the longer the effective service time is, the smaller the proportion of overhead brought from tracing thread is. Furthermore, we observe that the relative error of the curve indicating 10 loops is higher than that of the curves indicating 30, 50, 70 and 90 loops, and the curves indicating 50, 70, 90 and 110 loops are consistent. When it loops for more times, the error rate falls to about 1 percent. So our method has high accuracy when the system runs for a long time, because the stochastic error is canceled out.

For evaluating the accuracy of the memory monitoring method, we deployed a service bundle which implements a service to create an integer array of 100,000 elements. Thus we referred the array to an instance of a class, lest it should be garbage collected by the JVM. The controller bundle invokes the service for 10 times, and sleeps for 5 minutes after every invocation. So it is obvious that the memory utilization of the service bundle increases by about 0.4 Mbytes every 5 minutes because of service in-vocation for memory allocation. As is shown in Figure 7, the observed curve of this experiment is

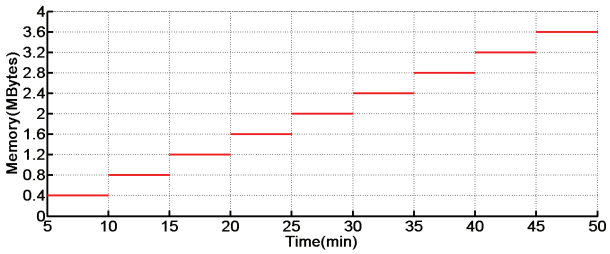

Fig. 7: Memory monitoring accuracy

consistent with the expected result. The accuracy of monitoring bundle memory utilization is higher than 99.8 percent.

5.2 Performance and resource overhead

In this part, we apply our method in a real application server OnceAS [13], which has been transformed to the OSGi framework [14]. The overhead introduced by our method is evaluated in this subsection. The overhead is considered from two perspectives that are performance metrics including average response time and throughput, and the resource utilization including CPU time and heap memory.

Table 1: Testbed components

Component	Processor	RAM
Application Server(OnceAS)	Intel Xeon 2.5GHz(8 CPUs)	2G
Database(DB2)	Intel Xeon 3.0GHz(4 CPUs)	2G
Clients(Emulated Browsers)	Intel Core 2 Duo 2.33GHz(2 CPUs)	2G

In our experiments, we use a testbed of a standard three-tier e-commerce application, and simulate the operations of an online bookstore, according to TPC-W specification [15]. Specifics of the software/hardware are given in Table 1. The client's access to the web site occurs as a session consisting of a sequence of consecutive individual requests. Users log in to the Website, browse the products, add several books into the shopping cart, check out the order and log out of the website.

We simulate 25 to 350 concurrent browsers with different threads. The performance metrics evaluated for this scenario are the throughput that is the number of completed transactions per second, and average response time that is the time taken to complete a transaction. As is shown in Figure 8, from the comparison of performance metrics, we can see that the performance of OnceAS with and without monitoring is considered equivalent. When

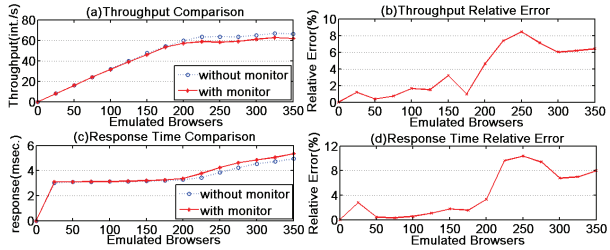

Fig. 8: Performance overhead

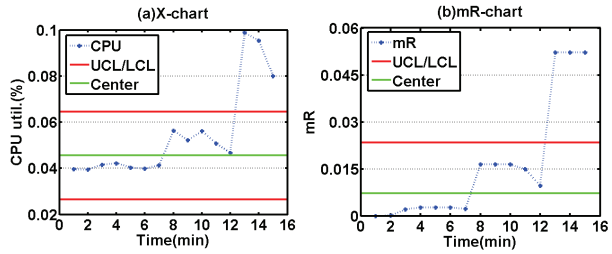

Fig. 10: Control chart for CPU intensive loop

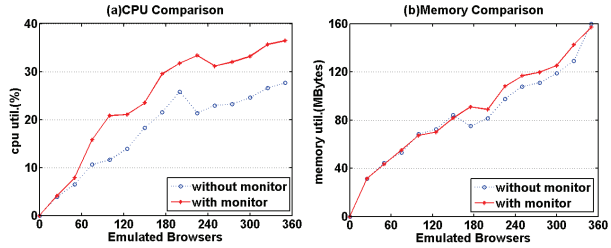

Fig. 9: Resource overhead

the number of concurrent users is less than 175, the system is not saturated, so its throughput increases and response time keeps about 3.2 seconds as the number of users grows. After that point, the system becomes saturated, so its throughput does not increase anymore and the response time increases. In order to understand the overhead brought from monitoring clearly, the performance overhead is studied in terms of relative error which is calculated with the following formula.

$$RE = |v - v_{approx}|/v, \tag{8}$$

where v represents the performance metrics without monitoring, and v_{approx} represents the performance metrics with monitoring.

The performance overhead is less than 3.2 percent when the number of browsers does not exceed 175, and less than 10.3 percent after that point, so the performance overhead brought from monitoring is not significant.

This is explained by the low resource overhead. As is shown in Figure 9, the CPU utilization of the system with monitoring is about 8 percent more than that of the system without monitoring. The overhead is caused by tagging threads and objects, and tracing threads with the JVMTI. At the same time, the memory utilization of the system with monitoring is about 9M bytes more than that of the system without monitoring. The overhead is caused by additional service proxy objects. From the above results, we can see our method without significant overhead is applicable in the real deployment environment.

5.3 Effectiveness of detecting anomalous components

To validate our method for detecting anomalous components, we inject two typical faults in the HTTP service bundle which is responsible for parsing HTTP requests in OnceAS. Since injecting faults is a difficult issue which is out of our scope, we choose two typical real faults as analyzed in [16], and inject them with the method used in [17, 18].

In the experiment, we also use the testbed. We simulate 300 concurrent users from 1st to 8th minute, and 400 concurrent users from 9th to 15th minute. Each experiment lasts 15 minutes, the injected faults are triggered in the HTTP service in the 12th minute through timing automatically, and we monitor system status every minute. As is shown in Figure 10 and Figure 11, the x-axis represents sampling time, the y-axis in (a)X-chart represents resource utilization per-interaction, and the y-axis in (b)mR-chart represents the moving range. The results show that individual measurements and moving ranges are in the normal scales before the faults are injected. Nevertheless, some anomalies are detected after the following faults are injected.

CPU intensive loop: results from circular wait or endless loop in program such as spin lock fault. We inject it by inserting the additional computation operation which is a loop for 5ms. In each interaction with the injected service, these operations are triggered to consume additional CPU time. As is shown in Figure 10 which describes the XmR control charts of CPU utilization, after injecting this fault, the individual measurements and moving ranges are both higher than UCL from the 13th minute. Thus we detect that some anomalies occur in the HTTP service, and they are related to CPU processor.

Memory leak: is caused by locating heap memory to objects without releasing, so that it leads to the system crash eventually. In each interaction with the injected HTTP service, an object with the size of 10K bytes is created and referred to a static variable lest it should be garbage collected by the JVM. As is shown in Figure 11 which describes the XmR control charts of memory utilization, after injecting this fault, we detect anomalies from the 14th minute in the X-chart, and in the 15th

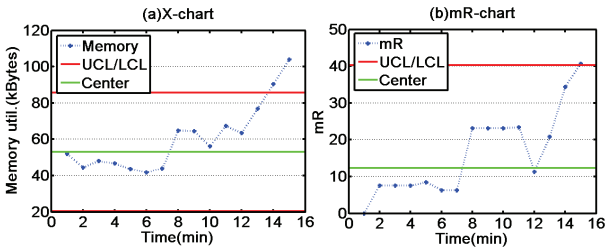

Fig. 11: Control chart for memory leak

minute in the mR-chart. Since the memory leak gradually exhausts the system memory resource, it is difficult for us to detect immediately. As is shown in the above experiments, our method can effectively detect anomalous services. Note that the faults in our experiments are only examples, and our method is also applicable to select other metrics and detect other faults which affect these metrics. Although we cannot locate the root cause of a fault in a line of code, operators can follow these anomalous metrics to narrow down possible causes. In addition, since OSGi provides a hot plug-and-play mechanism for services, when an anomalous service is detected, it is convenient for us to rejuvenate the service through re-installing or replace it with another one at runtime.

6 Discussion and future work

The accuracy is an important factor for any monitoring tool. We use JVMTI, which is a naive code based method, to calculate the CPU utilization of every thread. The native agent probes CPU for the calculation of cycles by sampling. The accuracy of our method is subject to the CPU resolution time of the operation system. It is impossible for us to improve the absolute precision defined by the CPU resolution time. For example, that of the Windows XP is 15.625 milliseconds, so our method cannot exceed that if our monitoring tool is deployed on the Windows XP. In the future work, we plan to use a statistical method such as the Kalman filter to correct the monitored data. Furthermore, we can also use some platform-specific tools to improve the precision.

The overhead is an obstacle to the application of a monitoring tool. Tagging object introduces a major significant overhead in our method. If threads are created and destroyed, or components interact with each other frequently, lots of thread objects ought to be tagged. There will be lots of calls to the monitoring agent function, and these calls of the naive codes are much slower than the Java method calls, so great overhead is introduced. In the future work, we plan to use a dynamic map to record the relationship between Bundle IDs and Thread IDs for reducing the traps in the naive code to a minimum.

Although our method can detect anomalous components effectively, we cannot locate the root cause of a fault in a line of code, and operators ought to follow many other anomalous metrics to narrow down the possible causes. In the future work, we plan to extend our method to collect other metrics for fine-grain fault location. Furthermore, since OSGi provides a hot plug-and-play mechanism for components, we will implement a framework to rejuvenate the component through re-installing it or replace it with another one automatically, when an anomalous component is detected.

7 Related work

7.1 Java application resource monitoring

Prevailing methods to measure CPU consumption in Java application mainly rely on native code libraries, which probe CPU for calculating cycles by sampling. For example, Magpie used Event Tracing with the processor cycle counter in Windows operating system [19]. Similar methods on other operating systems include the Linux Trace Toolkit [20] and Solaris DTrace [21], etc. These methods, which instrument applications at the source or binary level, rely on the operating system kernel to collect the events. Binder et al. proposed a portable CPU-management framework for Java, which tracked the number of executed the JVM bytecode instructions, and then transformed them to CPU consumption [22]. These methods introduce significant overhead at runtime. Furthermore, they all aim at the whole JVM instead of services or components. We transform the resource perspective to service-oriented component level in the OSGi framework.

The most related work to ours was conducted by Miettinen et al., which created a unique ThreadGroup object for every bundle deployed to OSGi [9]. The task executed by one thread in the original software is executed by different threads belonging to different ThreadGroups sequentially. However, this method modifies the thread schedule model. Moreover, complex thread scheduling mechanism, frequent thread switching operations and maintenance of a large number of threads bring significant overhead. Therefore, this OSGi-based monitoring method is only suitable for off-line simulation test, but not applicable in the real deployment environment.

7.2 Anomaly detection

Commercial monitoring tools are widely used to detect anomalies in practice, e.g., IBM Tivoli, HP OpenView. System operators manually set rules to collect monitoring data and trigger alerts with these tools. When the metric exceeds its defined threshold, some alerts are generated

automatically. However, it is difficult to set suitable thresholds for so many metrics in complex component-based systems. Signature based methods define the signatures of known faults, and detect anomalies by matching a specific set of rules. Chen et al. stored historical failures and retrieved similar instances in the occurrence of failure [23]. The failure characteristics were described as an invariant network. Ghanbari et al. used Bayesian networks to learn fault symptoms from labeled data [24]. These methods are effective when the signatures of faults are well defined. However, it is difficult to recognize unknown faults.

Many studies model the system behaviors including execution paths and component interactions. Chen et al. used a probabilistic context-free grammar to represent the execution paths, in which grammar symbols were components used in servicing requests, and grammar rules corresponded to transitions assigned probabilities between components. The paths which failed to be parsed by grammar were regarded as anomalies [25]. Barham et al. used clustering to group paths, and the ones which did not fit the built clusters were anomalous [19]. Chen et al. employed statistics to periodically analyze interaction between one component and the others using x^2-test [26]. These methods are capable of detecting application-level faults. However, they cannot detect the anomalies caused by component resource utilization.

Metric correlation based methods characterize the hidden invariant relationships among system metrics, and the anomalies are detected when the relationships are broken. Jiang et al. used autoregressive linear regression with exogenous input (ARX) models to capture the metric correlations [17], and discussed two algorithms to speed up the discovery of metric correlations [27]. Munawar et al. discussed many linear regression methods to discover metric correlations [28]. Guo et al. investigated Gaussian Mixture Models (GMM) to model the nonlinear correlations between metrics [18]. While the methods are easy to be extended to many applications without domain specific knowledge, it is difficult to model various correlations between so many metrics in complex systems, and the metric correlation changes as workload pattern evolves [29]. Furthermore, these methods take the whole application as target, so it is not applicable for locating specific anomalous components in the component-based applications.

Some studies pay attention to performance anomaly detection. Cherkasova et al. proposed a regression-based model to reflect application resource consumption, and introduced an application performance signature to model the run-time application behavior. This work concentrates on CPU utilization regardless of other metrics [30]. Cohen et al. proposed TANs to identify which system-level metrics were correlated with high-level performance SLO (Service Level Object) violations [31]. The work aims at finding critical metrics which have an important impact on performance instead of tracking system status to detect anomalies. Gama et al. presented a self-healing sandbox for the execution of third party components in OSGi. In the sandbox, no faults are propagated to the trusted parts of the application [32]. The protocol between the trusted platform and the sandbox platform brings considerable performance overhead, and the correct functioning is based on a set of assumptions which may not apply to some real applications.

8 Conclusion

The OSGi framework provides support for the management of service-oriented applications. It is important for improving the reliability of OSGi-based applications to detect anomalies in the granularity of the service-oriented component. We propose a tracing-thread method for monitoring service-oriented components in OSGi-based applications. Our method, neither modifying software nor introducing significant overheads, is suitable for monitoring online. According to the monitored data, we further employ control charts to detect anomalous components. This method does not require prior knowledge and can detect the anomalous trend of component resource utilization. A prototype tool is designed and implemented in a real application server. The experimental results demonstrate our approach can monitor service resource utilization in high accuracy without significant overhead, and detect the anomalous services effectively.

Acknowledgement

This work is supported by the National Grand Fundamental Research Program of China (973) under Grant No. 2009CB320704, the National High Technology Research and Development Program of China (863) under Grant No. 2012AA011204, the National Natural Science Foundation of China (NSF) under Grant No. 61173004. The author is grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.
The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

References

[1] B. Councill, G. T. Heineman, Component-based software engineering and the issue of trust, in Proceedings of International Conference on Software Engineering, 661-664 (2000).
[2] D. Marples, P. Kriens, The Open Services Gateway Initiative: an introductory overview, IEEE Communications Magazine, **39**, 110-114 (2001).

[3] H. Cervantes,R. S. Hall, Autonomous adaptation to dynamic availability using a service-oriented component model, in Proceedings of the 26th International Conference on Software Engineering, 614-623 (2004).
[4] OSGi. About the OSGi Service Platform. Revision 4.1. Available: http://www.osgi.org/wiki/uploads/Links /OSGiTechnicalWhitePaper.pdf, (2007).
[5] M. R. Lyu, Software reliability engineering: a roadmap, in Proceedings of Future of Software Engineering, 153-170 (2007).
[6] V. Chandola, A. Banerjee,V. Kumar, Anomaly detection: A survey, ACM Computing Surveys, **41**, 1-58 (2009).
[7] S. S. Gokhale, Architecture-based software reliability analysis: Overview and limitations, IEEE Transactions on Dependable and Secure Computing, **4**, 32-40 (2007).
[8] H. Koziolek, Performance evaluation of component-based software systems: A survey, Performance Evaluation, **67**, 634-658 (2010).
[9] T. Miettinen, D. Pakkala, M. Hongisto, A method for the resource monitoring of OSGi-based software components, in Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications, 100-107 (2008).
[10] G. Kiczales, J. Lamping, A. Mendhekar, et al., Aspect-oriented programming, in Proceedings of the 15th European Conference on Object-Oriented Programming, 220-242, (1997).
[11] D. Jeffrey, G. David, C. Craig, Optimization of object-oriented programs using static class hierarchy analysis, in Proceedings of the 9th European Conference on Object-Oriented Programming, Aarhus, Denmark, 77-101 (1995).
[12] G. A. Barnard, Control charts and stochastic processes, Journal of the Royal Statistical Society, **21**, 239-271 (1959).
[13] W. Zhang, B. Yang, B. Jin, et al., Performance Tuning for Application Server OnceAS, in Proceedings of International Conference on Parallel and Distributed Processing and Applications, 451-462 (2005).
[14] T. Wang, X. Zhou, J. Wei, et al., Towards runtime plug-and-play software, in Proceedings of the 10th International Conference on Quality Software, 365-368 (2010).
[15] D. A. Menasc, TPC-W: A benchmark for e-commerce, IEEE Internet Computing, **6**, 83-87 (2002).
[16] S. Pertet,P. Narasimhan, Causes of failure in web applications, Parallel Data Laboratory, Carnegie Mellon University, CMU-PDL, 05-109 (2005).
[17] G. Jiang, H. Chen,K. Yoshihira, Modeling and tracking of transaction flow dynamics for fault detection in complex systems, IEEE Transactions on Dependable and Secure Computing, **3**, 312-326 (2006).
[18] G. Zhen, G. Jiang, H. Chen, et al., Tracking probabilistic correlation of monitoring data for fault detection in complex systems, in Proceedings of International Conference on Dependable Systems and Networks, PA, USA, 259-268 (2006).
[19] P. Barham, A. Donnelly, R. Isaacs, et al., Using magpie for request extraction and workload modelling, in Proceedings of the 6th International Symposium on Opearting Systems Design and Implementation, California, USA, 18-31 (2004).
[20] K. Yaghmour, M. R. Dagenais, Measuring and characterizing system behavior using kernel-level event logging, in Proceedings of the USENIX Annual Technical Conference, San Diego, California, 2-15 (2000).
[21] B. M. Cantrill, M. W. Shapiro, A. H. Leventhal, Dynamic instrumentation of production systems, in Proceedings of USENIX Annual Technical Conference, 2-15 (2004).
[22] J. Hulaas, W. Binder, Program transformations for portable CPU accounting and control in Java, in ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, Verona, Italy, 169-177 (2004).
[23] H. Chen, G. Jiang, K. Yoshihira, et al., Invariants based failure diagnosis in distributed computing systems, in Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems, India, 160-166 (2010).
[24] S. Ghanbari, C. Amza, Semantic-driven model composition for accurate anomaly diagnosis, in Proceedings of International Conference on Autonomic Computing, Illinois, USA, 35-44 (2008).
[25] M. Y. Chen, A. Accardi, E. Kiciman, et al., Path-based faliure and evolution management, in Proceedings of the 1st Symposium on Networked Systems Design and Implementation, Berkeley, CA, 23-36 (2004).
[26] H. Chen, G. Jiang, C. Ungureanu, et al., Failure detection and localization in component based systems by online tracking, in Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, Chicago, Illinois, USA, 750-755 (2005).
[27] G. Jiang, H. Chen,K. Yoshihira, Efficient and scalable algorithms for inferring likely invariants in distributed systems, IEEE Transactions on Knowledge and Data Engineering, **19**, 1508-1523 (2007).
[28] M. A. Munawar, P. A. S. Ward, A comparative study of pairwise regression techniques for problem determination, in Proceedings of Conference of the Center for Advanced Studies on Collaborative Research, Toronto, Canada, 152-166 (2007).
[29] M. Jiang, M. A. Munawar, T. Reidemeister, et al., System monitoring with metric-correlation models: problems and solutions, in Proceedings of the 6th International Conference on Autonomic Computing, Barcelona, Spain, 13-22 (2009).
[30] L. Cherkasova, K. Ozonat, N. Mi, et al., Automated anomaly detection and performance modeling of enterprise applications, ACM Transactions on Computer Systems (TOCS), **27**, 1-32 (2009).
[31] I. Cohen, M. Goldszmidt, T. Kelly, et al., Correlating instrumentation data to system states: a building block for automated diagnosis and control, in Proceedings of the 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA, 16-29 (2004).
[32] K. Gama,D. Donsez, A Self-healing Component Sandbox for Untrustworthy Third Party Code Execution, Component-Based Software Engineering, **6092**, 130-149 (2010).

Tao Wang is now a PhD candidate in the Institute of Software, Chinese Academy of Sciences, China. He received the MS degree in Computer Science from the University of Electronic Science and Technology of China, China in 2008. His research interests include fault detection, software reliability, and autonomic computing.

Jun Wei is a professor in the Institute of Software, Chinese Academy of Sciences, China. He received his BSc and PhD in Computer Science from the Wuhan University, Hubei, China in 1992 and 1997, respectively. He was a Postdoctoral Researcher at the Hong Kong University of Science and Technology, China. His research interests include service oriented computing, middleware, and software engineering.

Wenbo Zhang is an associate professor in the Institute of Software, Chinese Academy of Sciences, China. He received his PhD from the Institute of Software, Chinese Academy of Sciences, China in 2007, and his MEng in Computer Science and Technology from Shandong University in 2002. His research interests include service oriented computing and middleware.

Hua Zhong is a professor in the Institute of Software, Chinese Academy of Sciences, China. He received his PhD in Computer Science from the Institute of Software, Chinese Academy of Sciences in 1999. His research interests include software engineering and distributed computing.