A Trace Agent with Code No-invasion Based on Byte Code Enhancement Technology

Hongrun Wang
School of Computer Science
Beijing University of Posts and Telecommunications
Beijing 100876, China
rainrun @bupt.edu.cn

Wei Fang
School of Computer Science
Beijing University of Posts and Telecommunications
Beijing 100876, China
fang_wei @bupt.edu.cn

Abstract—**With the popularity of microservice architecture and the increasing complexity of services, how to effectively monitor application performance becomes more and more important. Under the guidance of the Dapper system of Google, many distributed tracking systems have emerged. The most essential part of the distributed tracking system is how to access the information of trace. In this paper, a trace agent based on byte code enhancement technology is presented. Then, the design of the agent and the construction of the trace logic are described in detail. The availability of agents is verified from the aspects of overhead and effectiveness under real environment. Finally, the result of experiment proves that it is convenient and effective to use the trace agent to get the information of trace from applications in Java language.**

Keywords-microservice architecture; distributed tracking systems; trace agent; byte code enhancement

I. INTRODUCTION

With the popularity of micro-service architectures and the increasing complexity of services, services are split according to different dimensions. One request often involves multiple services [1]. Applications are built on different modules that may be developed by different teams and programming languages. And these modules may be deployed on thousands of servers. Therefore, some tools to analyze application performance are necessary. When the application fails, they could locate and resolve the issue quickly.

In 2010, Google published Dapper, a large-scale distributed system tracing infrastructure [2]. Under the guidance of this paper, many application performance management [3] came into being gradually. In order to understand the business event at the micro-service architecture, it is necessary to monitor the associated actions directly across different modules and servers. Therefore, almost every front-end request forms a complex distributed service trace in a complex distributed system.

How to catch trace logs is an important part of building distributed tracking system. Trace logs are the context of the current node. The logs usually contain traceId, spanId, call start time, protocol type and so on. The traditional method to acquire trace logs refers to code invasion ways. The one is SDK that provides application development. The other is framework which provides developers code for catching trace logs. This paper proposes a trace agent based on byte code enhancement technology. The agent can acquire trace logs with code no-invasion. When starting the service modules, there needs to add some parameters. In this way, user code could gather trace logs transparently.

The remainder of this paper is structured as follows. In section II, we summarize the related work about trace agent. Then, we introduce the execution process and design of trace agent in section III. In addition, the trace data structure and construction will be also presented. In section IV, we apply this agent in real environment to verify the availability of agent form the aspects of overhead and effectiveness. Finally, the paper makes a summary and discusses the future work in section V.

II. RELATED WORK

According to the mentioned situations, it is necessary for acquiring trace logs. We briefly review the relevant literature.

Anderson E presents the challenges of distributed systems and the necessity of trace [5]. Rodrigo Fonseca proposes X-Trace, a tracing framework that provides such a comprehensive view for systems that adopt it [6]. Jingwen Zhou presents MTracer which is a lightweight trace-oriented monitoring system for medium-scale distributed systems [7]. Jonathan Mace presents Pivot Tracing, a monitoring framework for distributed systems that addresses both limitations by combining dynamic instrumentation with a novel relational operator: the happened-before join [8]. All of them desigined and implemented the distributed tracking system.

Zipkin is an open source distributed real-time data tracking system [9]. Karumuri S. presents PinTrace. It is Zipkin based distributed tracing infrastructure [10]. It can be accessed by introducing a tool library for capturing and reporting delay information of distributed operations in the business code. This is a typical code invasion method. It has good scalability. We can develop SDKs for different languages. However, in the scenario of application upgrade, the code needs to be remodified. The maintenance cost is extremely high. At the same time, it is difficult for developers to accurately identify the location of the buried point. In this way, the tracking level is low and it is impossible to obtain enough detailed dynamic information.

978-1-7281-0945-9/19/$31.00©2019 IEEE

M Rasmussen uses a method to dynamically modify byte code when the JVM is loading classes [11]. F. Freitag propose a trace-scaling agent for tracing parallel applications [12]. Da Silva describes JRastro, a trace agent capable of tracing Java programs [13]. Both of them used agent to trace applications. But they can't solve the problem of trace construction.

III. TRACE AGENT ARCHITECTURE

In this section, we firstly introduce the execution process of the trace agent. Then, we will introduce the initialization of the trace agent and the interception logic of the internal plugin. Finally, we discuss the trace structure and construction logic.

A. Trace Agent Implementation Process

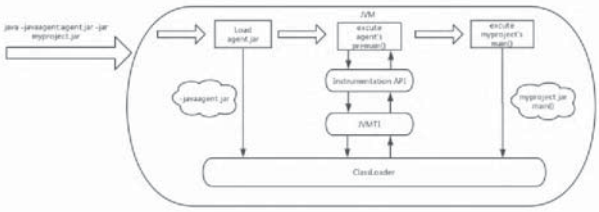

Figure 1. Trace agent implementation process

In Figure 1, we show in detail the execution flow of a byte code enhancement agent. Use the *-javaagent* parameter when executing the jar package of the business module, and take the path of the *agent.jar* with the parameters to enter the *JVM* processing flow. There are three processes as follows:

a) Use the class loader to load the jar of the agent, each time the newly loaded class is enhanced by the code of the agent.

b) Execute the premain method of the agent. In the code of the agent, you need to use the *Instrumentation API*. We should call the *addTransformer* method to load into the *JVMTI*. Then, we listen to the class file to read the event. The *ClassLoader* needs to call back before each class is loaded.

c) Execute the main method of the program which is the code of business module after executing the *premain* method.

B. Trace Agent Design

TABLE I. PARAMETER TABLE

Parameter	Description
C_I	Use for initialize agent configuration
P_B	Use for load agent plugins
S_M	Use for initialize agent service management
A_G	Provide convenient APIs to create Java agent
P_{DS}	A list contains P_Ds
P_D	Use for define aspect, record the call chain
Bu	Define how to intercept modified Java classes

```
1: function PREMAIN(agentArgs,instrumentation)
2:     C_I.initialize()
3:     P_B.loadPlugins()
4:     S_M.boot()
5:     A_G.type().transform()
6: end function
7: function TRANSFORM(name)
8:     P_Ds ← find(name)
9:     for all P_D : P_Ds do
10:        Bu ←P_D.define(name)
11:        if Bu.exist() then
12:            return Bu
13:        else
14:            break
15:        end if
16:    end for
17: end function
```

Algorithm 1. The logic of agent initialization

Algorithm 1 explains the execution process of the *premain* method initialization. The meaning of the parameters is shown in the table I. Firstly, we call the *initialize* method to initialize the configuration of the Agent, then the *loadPlugin* method loads the Agent plugin. When it comes to the boot method, it initializes the Agent service management. Finally, we use the type method of the *AgentBuilder* class to set the intercepted class, and call the *transform* method to set the modification logic of the Java class. In the *transform* method, we should find all the plugin definition arrays matching the name by the *find* method. When traversing the plugin definition array, we are setting the Builder object through the *define* method which defines how to intercept the Java classes that need to be modified. Inside the *define* method, we use the *intercept* method of the *bytebuddy* package to set the interception.

The above process refers to the plugin system of Agent, which consists of three parts:

a) Plugin loading. When the Agent initializes, it calls the *loadPlugin* method to load all the plugins. Within the method, we initialize *AgentClassLoad* and get the plugin definition array. Then we create a class enhancement plugin definition object array. Different plugins define the aspects of the different frameworks by implementing the same abstract class, This can help us record the trace.

b) Plugin matching. We are using a matching rule based on the full class name. By matching the class name, we can find the corresponding *PluginDefine* object.

c) Plugin intercepting. Intercepting contains InterceptPoint, Interceptor and Define. Define refers to which interception aspect and corresponding interceptor of a class. In Figure 2, we describe the process of plugin interception.

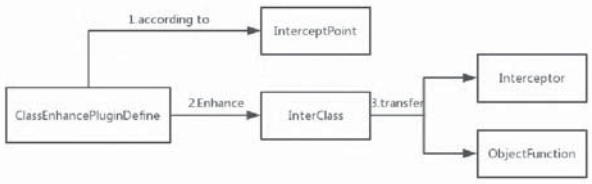

Figure 2. Plugin intercepting process in agent

We use an example to better understand the meaning of Figure 2, we want to design a *SpringMVC* plugin. Firstly, we need to design a *SpringMVCDefine* class, in which the intercept point of the constructor, static methods and instance methods. These intercept points can find the entry of the method when loading the binary code. we design the interceptor, which can perform some acquisition of the trace information from these methods. For example, when we are calling an interface, service name, method name, current time and other information of the method are intercepted. In summary, the intercept point is defined according to the *PluginDefine* class. An instance class is enhanced and the interceptor can intercept the method called in the instance class.

C. Trace Data Structure and Construction

Before we start talking about this section, we must first understand *OpenTracing* [14], which is a data model involved in the tracking system. It defines a trace that can be considered to consist of multiple spans. These spans can form a directed acyclic graph. A trace contains a lot of modules and call relationships between modules. The agent we described above intercepts some information, such as start/end timestamp, node address and service type by plugins. We will introduce the data structure of the trace information collected and the logic of how to build the trace.

TABLE II. TRACE DATA STRUCTURE

Name	Description
traceId	Recording call chain unique identifier
rpcId	Recording node path relation
traceName	Recording request uri
invokerName	Recording request upstream node service name
serviceName	Recording request service name
methodName	Recording request method name
rpcType	Recording RPC type
resultType	Recording result type
isRoot	Recording whether current node is root or not
startTime	Recording request start time
endTime	Recording request end time
ip	Recording IP address
pid	Recording process id
logs	Recording additional information
product	Recording product name
url	Recording request url

Table II shows the data structure of the trace. Then we discuss how the agent generates this data. We define the *RpcContext* class that contains the trace data structure. There is a counter inside to guarantee the order of the *RPC* request. Inside the main program, when there are multiple child threads processing the request, the child thread inherits *RpcContextThreadLocal* of current parent. The *RpcContextThreadLocal* of the thread ensures that the information of parent thread can interact with the child thread.

```
1: function BEFOREMETHOD
2:     if isNewTrace then
3:         clientSend()
4:     else
5:         serverRecv()
6:     end if
7: end function
8: function CLIENTSEND
9:     if hasParentContext() then
10:        childContext ← createChildContext()
11:    else
12:        startTrace()
13:    end if
14: end function
15: function SERVERRECV
16:    if validate(traceId) then
17:        startTrace()
18:        send()
19:    else
20:        break
21:    end if
22: end function
```

Algorithm 2. The logic of trace construction

Algorithm 2 shows that the starting trace from the plugin interception. we need the interceptor to intercept a method, and then determine whether it is a new trace. if it is a new trace, it will execute the *clientSend* method. The method sends an *RPC* call downstream. The current thread needs to have *rpcContext*, otherwise a trace will be created. if not, it will execute the *serverRecv* method, which receives the *RPC* call sent upstream and returns the processing result to the upstream after execution. Finally, it releases the current *rpcContext*. After the algorithm is executed, the construction of the trace will be completed. It will involve a *traceId* how to ensure consistency in a distributed system. For instance, we use *HTTP* to initiate a call between the module and the module. We can put the *traceId* in the *HTTP* header to ensure that the submodule can receive the *traceId*. If an *RPC* call is used, the *traceId* can be serialized to the submodule through the serialization protocol to ensure consistency. The trace data we collected will be sent to the collector of the tracking system.

IV. EXPERMENT AND RESULT

It is a trend that using agent with no-invasion to access business module. The business modules can focus on the development of business logic. They don't want to spend too much manpower adapting to performance monitoring platform. In this part, we validate trace agent from two aspects: overhead

and effectiveness. Table III shows the machines used in the experiment.

TABLE III. EXPERIMENTAL ENVIRONMENT

Index	Value
CPU	Inter® Core™ i5-8300H CPU @ 2.30GHz
Memory	8GB
OS	Windows 10 x64

A. *Overhead*

We use the modules developed by Spring Boot to test the performance of using agent startup. The experimental results are shown in Table IV.

TABLE IV. EXPERIMENTAL RESULT

	Use agent	Don't use
Starting up time	18.33s	15.128s
JVM run time	21.691s	17.734s
Avg Thread	38	34
Max Thread	42	37
Submit Memory	573216KB	419840KB
Usage Memory	126443KB	95796KB

Using the agent to start the service will inevitably bring some overhead, because we need to obtain some monitoring information through the agent. We compare the analysis from the application startup time, the average number of threads, the maximum number of threads, the committed memory size and the memory size. It can be seen that the use of the agent will bring a certain resource occupation. A comparative analysis shows that the use of agents has some of loss in both memory usage and startup time. We consider distributed scenarios at this time. If you want to analyze the application, this loss is acceptable. We also analyzed the interception of a request. The average event for intercepting a request was 0.089 milliseconds. In a distributed system, many operations take a few seconds or even minutes to complete, so this time consuming is acceptable.

B. *Effectiveness*

In this part, we consider this method from the validity. We package our service into *device-management.jar* and start the jar through *java -javaagent:[agent.jar path]/agent.jar -jar device-management.jar*. Then, we can see some information about the service through the UI.

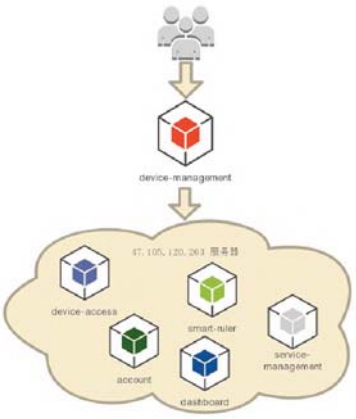

Figure 3. The topology of module

There are many interfaces for user calls in device-management. As the result shown in Figure 3, the user can call one of interface in device-management. The response may come from results of different modules. Time-consuming, calling modules and attribute structure of each interface can be shown in UI. In Figure 4, we show the tree structure of an interface. The response of */api/v1/abilityGroup* comes from three request in different modules.

Figure 4. The tree of request chain

V. CONCLUSIONS

A byte code enhanced method for the tracking system was proposed in this paper. The byte code enhanced method can catch function level trace information. It can collect the calling information of the stack level without modifying the code. Therefore, the tracking level is high. We design the trace agent based on byte code enhancement. The service application uses the trace agent to access the tracking system. Based on the agent architecture design and trace logic, the paper introduces the algorithms involved in the agent in detail. Then we use the agent to conduct experiments to prove the availability of method from the load and effectiveness. The use of byte code enhanced technology obtaining trace information allows the service provider to transparently access the tracking system. It can use the system of application performance analysis quickly. And it can provide powerful help for building safe and stable applications. But the agent mentioned in this article is only suitable for the embedding of application modules developed in Java language. In the future work, we will be dedicated to develop trace agent to access applications in different languages.

REFERENCES

[1] Balalaie A , Heydarnoori A , Jamshidi P . Microservices Architecture Enables DevOps: an Experience Report on Migration to a Cloud-Native Architecture[J]. IEEE Software, 2016:1-1.

[2] Sigelman B H, Barroso L A, Burrows M, et al. Dapper, a large-scale distributed systems tracing infrastructure[J]. 2010.

[3] Ahmed T M, Bezemer C P, Chen T H, et al. Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications: an experience report[C]// Proceedings of the 13th International Conference on Mining Software Repositories. ACM, 2016: 1-12.

[4] Karumuri S. Pintrace: A Distributed Tracing Pipeline[J]. 2017.

[5] Anderson E , Hoover C , Li X , et al. Efficient tracing and performance analysis for large distributed systems[C]//IEEE International Symposium on Modeling. IEEE, 2009.

[6] Mace J, Roelke R, Fonseca R. Pivot tracing: Dynamic causal monitoring for distributed systems[J]. ACM Transactions on Computer Systems (TOCS), 2018, 35(4): 11.

[7] Zhou J, Chen Z, Mi H, et al. MTracer: A Trace-Oriented Monitoring Framework for Medium-Scale Distributed Systems[C] IEEE International Symposium on Service Oriented System Engineering. 2014.

[8] Mace J, Roelke R, Fonseca R. Pivot tracing: Dynamic causal monitoring for distributed systems[J]. ACM Transactions on Computer Systems (TOCS), 2018, 35(4): 11.

[9] Twitter, "Zipkin," http://twitter.github.com/zipkin, 2013.

[10] Karumuri S. Pintrace: A Distributed Tracing Pipeline[J]. 2017.

[11] Rasmussen M, Gregersen A R, Jorgensen B N. Method for dynamically transforming the bytecode of Java virtual machine bootstrap classes: U.S. Patent 9,141,415[P]. 2015-9-22.

[12] Freitag F , Caubet J , Labarta J . A trace-scaling agent for parallel application tracing[C]// 14th IEEE International Conference on Tools with Artificial Intelligence, 2002. (ICTAI 2002). Proceedings. IEEE, 2003.

[13] Da Silva G J , Schnorr L M , Benhur D O S . JRastro: a trace agent for debugging multithreaded and distributed Java programs[M]. 2003.

[14] OpenTracing: A vendor-neutral open standard for distributed tracing. http://opentracing.io/,2016