JCallGraph: Tracing Microservices in Very Large Scale Container Cloud Platforms

Haifeng Liu[1,2(✉)], Jinjun Zhang[1], Huasong Shan[1], Min Li[1], Yuan Chen[1], Xiaofeng He[1], and Xiaowei Li[1]

[1] JD.com, Beijing, China
[2] University of Science and Technology of China, Hefei, China
{bjliuhaifeng,zhangjinjun1,huasong.shan,min.li,yuan.chen, hexiaofeng,lixiaowei1}@jd.com

Abstract. Microservice architecture splits giant and complex enterprise applications into fine-grained microservices, promoting agile development, integration, delivery and deployment. However, monitoring tens of thousands of microservices is extremely challenging, and debugging problems among massive microservices is like looking for a needle in a haystack. We present JCallGraph, a tracing and analytics tool to capture and visualize the microservice invocation relationship of tens of thousands of microservices with millions of containers at JD.com. JCallGraph achieves three main goals for distributed tracing and debugging: online microservices invocation construction within milliseconds, minimal overhead without any significant performance impact on real-production applications, and application-agnostic with zero-intrusion to application. Our evaluation shows that JCallGraph can accurately capture the real-time invocation relationship at massive scale and help developers to efficiently understand interactions among microservices, pinpoint root-cause of problems.

Keywords: Microservice invocation graph · Distributed tracing system · Performance analysis and measurement

1 Introduction

Microservice architecture is increasingly embraced in enterprise in recent years due to its advantages and the advancement of container technologies [2–4,16,20,25]. Compared to monolithic architectures, microservice paradigm breaks up complicated enterprise applications and software into smaller, modular, independent components communicating through lightweight mechanisms such as a HTTP RESTful API. Since the components can be developed and maintained independently, it allows better development agility, isolation, resilience and scalability. However, as the number of services continue to grow in microservice platforms, the communication between components becomes complex resulting

D. Da Silva et al. (Eds.): CLOUD 2019, LNCS 11513, pp. 287–302, 2019.
https://doi.org/10.1007/978-3-030-23502-4_20

in high maintenance cost. In particular, it is difficult to identify root causes when errors occur or diagnose performance bottlenecks as the interactions between microservices are too complex to trace.

At JD.com [10], the world's third largest and China's largest e-commerce site, we provide more than 8000 applications and approximate 34,000 microservices run on the cluster of 500,000 containers, and support over 250 billions of RPC-based microservice calls per day. A microservice typically interacts with hundreds or even thousands of other microservices, not to mention that these services are often deployed and executed across a large number of containers or machines for performance and reliability. Moreover, the interactions change dynamically when services evolve and are developed independently. Thus, it is difficult to trace the communications and detect the updated system behavior whenever a component is changed.

Existing distributed tracing systems, such as Google's Dapper [24], Mace [11], Stardust [19], X-Trace [6], Retro [13], Pivot Tracing [14], can be used to monitor and trace the timing and interactions of system components. Yet, they cannot be directly applied into our container-based microservice platform that has millions of service instances. At JD.com, we particularly focus on the following requirements and challenges that enable our system to effectively trace microservices at massive scale.

Firstly, how do we provide microservice level transparency while automatically tracing all applications in the microservice platforms without active involvement of upper level services and applications? A tracing system that requires input from upper level components is more fragile and takes significantly more efforts to encourage the adoption. Inspired by Dapper [24], our system adopts an approach of limiting the intrusion within the underlying middleware such as JSF, JMQ and JIMDB.

Secondly, how do we effectively capture all the critical points in both physical and logical invocation chains? We want to minimize the intrusion of our tracing system into the existing microservice middleware while guaranteeing to capture interesting and critical points in execution paths. A *physical* invocation chain represents an actual execution path whereas a *logical* invocation chain refers to a business logic flow. One logical invocation chain usually contains one or more physical invocation chain. It is also important to minimize the overhead of the tracing system in our microservice framework. If the overhead is not negligible, the adoption of the system would be significantly impacted as users are likely to turn them off. Finally, the tracing system should be scalable to support tracing of all microservices that run on millions of containers.

In this paper, we present JCallGraph, a distributed tracing system that tracks the interactions and timing information of microservices across systems and machines. The logical invocation chains captured are particularly important to JD.com as it helps develop and maintain complex business logic. JCallGraph addresses the challenges by only intruding the underlying middleware of the microservice platform, and eliminating the need to intrude applications. JCallGraph also leverages sampling to dramatically reduce the overhead of the system. Efficient log transfer layer and in-memory storage layer are integrated to ensure

real time analysis and visualization. JCallGraph provides better understanding of the system such as analyzing complex execution paths across microservices. Other functionalities include timing and bottleneck analysis, analyzing statistics of calling information or the entrance points, and analyzing the dependencies within an invocation chain.

In particular, our paper has contributions as follows:

- Distributed dynamic tracing to provide a holistic view of both physical and logical invocation chains for large-scale microservice-based applications in an enterprise cloud platform with millions of containers.
- Careful intrusion of core middlewares to eliminate active involvement of applications; short UUID, low rate sampling and high compression context to increase the performance and scalability of the system.
- Comprehensive stress tests in test environment and deployment in real production environment, and use cases from our experiences, demonstrating the effectiveness and efficiency of JCallGraph.

We outline the rest of this paper as follows. Section 2 introduces microservices in action at JD.com. Section 3 describes the design and implementation of JCallGraph. Performance evaluation results and use cases are in Sect. 4. Section 5 presents the related work and Sect. 6 concludes the paper.

2 Microservices in Action at JD.com

The microservice architecture and container techniques facilitate software development, maintenance and delivery. So far we provide 34,000 microservices deployed in 500,000 containers managed by JDOS [26], a Kubernetes-based datacenter operating system. Such a huge number of microservices invoke each other to serve various business units, e.g., JD Retails [10], JD Finance [8], JD Logistics [9] etc. It is hard to observe and monitor the microservice invocation relationship in such a large-scale real-production environment. Figure 1 shows part of microservice invocation graph at JD.com. This actual demand motivated us to design JCallGraph: a monitoring system for tracing and visualizing the microservice invocation relationship.

To manage and support the development of large-scale microservices effectively and efficiently, we develop JSF, an RPC framework supporting; JMQ, a distributed messaging and streaming platform; JIMDB, a journaled in-memory database compatible with Redis [17]. As shown in Fig. 2, each application consists of multiple microservices. These applications synchronously invoke microservices through JSF, asynchronously communicate with each other through JMQ, and retrieve and store data in JIMDB. All of the applications work together to perform e-commerce or other transactions at JD.com.

JCallGraph serves as a tracing and visualization tool to show the microservice invocation graph among various applications. The objective of JCallGraph is to construct the microservice invocation relationship graph in the large-scale container environments on the fly, at the same time, it requires minimal impact to applications in real-production environment.

Fig. 1. Microservice graph at JD.com.

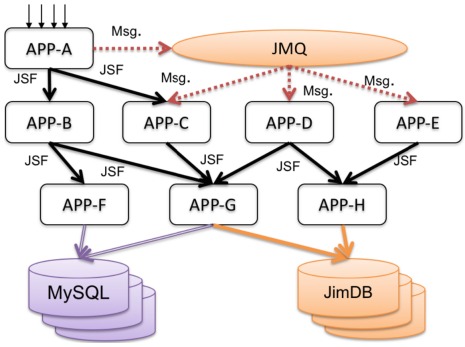

Fig. 2. An illustration of microservice invocation at JD.com. JSF as a synchronous framework, JMQ as an asynchronous messaging queue, JIMDB as a in-memory database.

3 Design and Implementation

3.1 Overview

To construct the invocation graph of microservices in the large-scale container environments, this greatest challenge is how to represent the invocation relationship, including the call flow and data flow. Previous work [24] uses end-to-end request flow tracing techniques to construct sequential and parallel activities in distributed systems. However, we have a unique requirement to fully comprehend the invocation relationship in our applications. For example, a customer purchases a product from JD Retails, pays through JD Finance, tracks the delivery using JD logistics. To monitor the performance of this business transaction, we need trace the real-time invocation graph among various microservices provided by several applications deployed across different machines and even across data centers. To this end, we not only need to construct the physical request-response flow as the previous work do, we also need to construct the logical microservice invocation graph, e.g., e-commercial order number.

Here, we use physical chain to trace the actual request flow among the applications, and logical chain to trace the business logic request chain for a special business requirement. All the logical and physical chains compose the complex microservice invocation graph at massive scale. Specifically, we design several identifiers (ID) to capture the microservice invocation graph. Specifically, we use two types of identifiers to represent the runtime invocation relationship for the applications and microservices: static identifiers (e.g., application ID, microservice ID) which are generated when the application and the microservice registered; and dynamic identifiers (e.g., global logical chain ID, RPC ID) which are generated during the runtime. In addition, since the invocation relationship is similar to a tree, we record the detailed structure of an invocation tree, such as the entry, parent, current node using these identifiers.

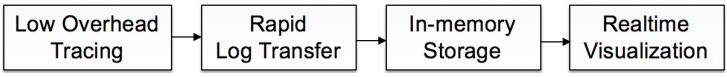

Fig. 3. Overview of JCallGraph's process pipeline.

Based on this idea, we design JCallGraph, including four components as shown in Fig. 3 (1) Trace Layer, tracing the invocation relationship in the middlewares (e.g., JSF, JMQ, JIMDB etc.); (2) Transfer Layer, transferring the traced invocation information to storages; (3) Storage Layer, storing the real-time data in JIMDB and offline analytic data in Elasticsearch[1] (4) Visualization and Analytics Layer, visualizing and deeply analyzing the microservices and their invocation relationship.

3.2 Tracing Microservices via Intruding upon Core Middlewares

JCallGraph's goal is to accurately construct microservice invocation graph at a lower cost without application intrusion. At JD.com, we use JSF, the microservice management platform to support over 250 billions of RPC-based microservice calls per day. To trace these invocation relationship, we add minimal critical tracing points in the core middlewares (e.g., JSF, JMQ, JIMDB), making zero code intrusion to thousands of applications and millions of microservices.

Primitives. JCallGraph's tracing layer provides several primitives to record the microservice invocation context, e.g., startTrace, endTrace, clientSend, serverRev, etc. We carefully place the primitives in the middlewares where it can record the request-response relationship among the microservices with minimal tracing points. Figure 4 depicts a concrete process to construct the invocation context using these primitives in the middlewares. Through the middleware (e.g., JSF, JMQ, JIMDB etc.), the front-end applications invoke startTrace to generate the global chain id and start a runtime invocation track; the upstream applications

[1] Elasticsearch. "https://www.elastic.co/".

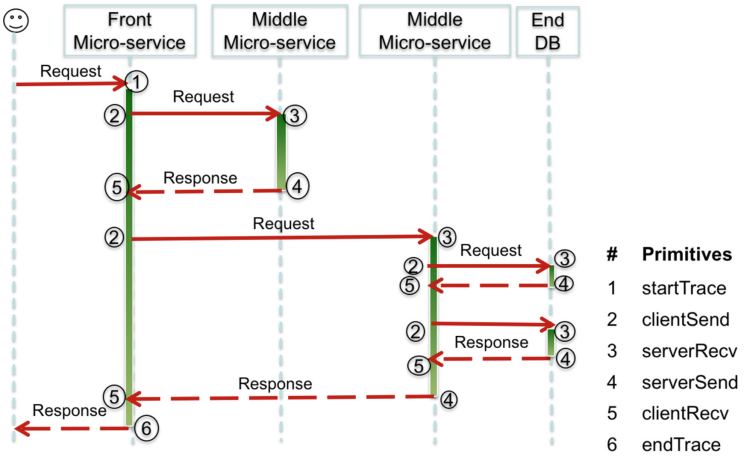

Fig. 4. A process to construct the invocation context using the primitives provided by JCallGraph. All the applications are unaware of the tracing process, since all the tracing points are in the middlewares.

invoke *clientSend* to record the start timestamp of the upstream application; once the downstream applications receive the request, they invoke *serverRecv* to record the start timestamp of the downstream application; after the downstream applications process the request, they invoke the *serverSend* to generate the downstream RPC id, record the end timestamp of the downstream application; once the upstream applications receive the response, they invoke *clientRecv* to generate the upstream RPC id and record the end timestamp of the upstream application; finally the front-end applications call *endTrace* to finish the specific runtime invocation track.

Tracing Invocations. The applications implement a RPC synchronous invocation by the interfaces provided by JSF. The invocation context is transferred during the process of the JSF, thus JCallGraph can restore the context of the synchronous invocation by plugging the primitives into the interfaces of JSF. For the asynchronous invocation, it involves the transparent transfer of the context among multi-threads. There are two cases as shown in Fig. 5: creating a new thread and requesting a thread from thread pools. JCallGraph intercepts the invocation context by adopting Java bytecode instrumentation technique [5] when JCallGraph's tracing primitives are called in the core middleware.

3.3 Low Overhead Tracing

To reduce the overhead in tracing layer and minimize the impact on the performance of applications, we adopt several specific techniques in the following.

Short UUID. We use unique identifier to record the entry, parent, current node in an invocation tree to record the runtime invocation relationship. Popular distributed systems such as OpenStack, Spark, and Hadoop use the universally

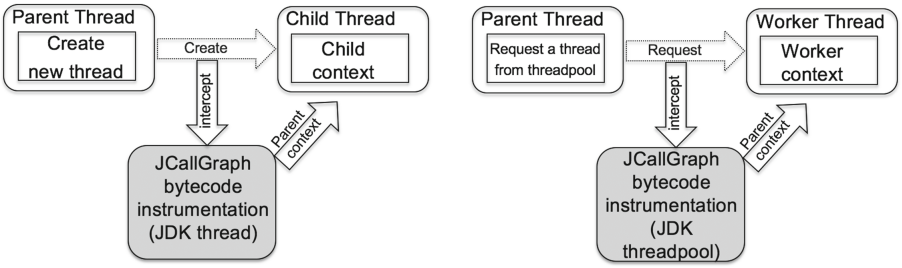

Fig. 5. Tracing the asynchronous invocation among multi-threads, including creating a new thread (left) and requesting a thread from the thread pool (right).

unique identifier (UUID), an 32 hexadecimal characters. However, 32 characters are too expensive for the tracing context, since it incurs more stress to the transfer and storage layer. Therefore, we use bit operations and cut down 32 characters into the 8 hexadecimal short UUID to uniquely represent these nodes of an invocation tree, which can dramatically reduce the overhead while still achieving similar low probability of conflicting identifiers as that by the traditional hexadecimal UUIDs with 32 bits. In our experiments, we observe that the likelihood of identifier conflict is one in a million, it is more than enough to distinguish all invocation chains since we can identify the invocation chain as long as all the RPC IDs in a runtime invocation chain is unique. Most importantly, the 8 hexadecimal short UUID can dramatically accelerate the tracing, transfer and visualization process.

Low Rate Sampling. Inspired by Dapper [24], to further reduce the impact on the applications, we use sampling. The difference is that we only sample successful invocation meanwhile recording all failure invocation. The benefit is that we can use very low sampling rate, but we still can guarantee the accuracy of constructing the normal invocation relationship. In our case the normal invocation is repeatable, once we miss some tracing, we can retry to guarantee the accuracy of constructing the invocation graph. We do not miss any failure invocation, which is useful for root-cause analysis. Various sampling rate impacts on the network traffic of transfer layer is available in Sect. 4.2.

All In-Memory. To eliminate the I/O contention between JCallGraph and the applications incurred by recording the call context, all the operations in tracing layer are in-memory, and we adopt an unlocked ring memory buffer to buff the traced context of the applications. Once the buffer is full, our policy is to drop these contexts to guarantee the efficiency of the applications, meanwhile we still can accurately construct the invocation graph through retrying.

3.4 Realtime Transfer and Visualization

We transfer the traced invocation information to underlying storage system through a transfer layer cluster, which consists of 16 nodes in real production environment, each node serves approximate thousands of application machines for tracing transfer through long keep-alive TCP connections. The real-time invocation information data are stored in the in-memory Database JIMDB and further offline analytic data are stored in Elasticsearch. JIMDB, as a cache layer, supports JCallGraph to visualize the online invocation graph of the microservices within milliseconds.

High Compression Context. In transfer layer, the main overhead comes from the additional network bandwidth consumption [18], which increases as the size of the messages increases which carry the information of invocation relationship. We represent the microservice invocation graph with the minimal context information. In particular, we effectively compress the invocation context by exploring the similarity between the contexts. In our cases, the same invocation chain typically shares the same context, and the invocation relationship is static for a specific application. Thus, we can achieve a very high compression ratio, usually 1/10 as observed in production environment. We observe that the actual message size is approximate 112 bytes for most of real-production applications, and using the message size can achieve the optimal throughput and resource utilization in our real-production environments. Detailed explanation can refer to our stress testing as shown in Sect. 4.2.

4 Evaluation and Experiences

4.1 Operation Data at JD.com

So far JCallGraph has been deployed in real-production environment for over than two years at JD.com, monitoring more than 8000 applications and approximate 34,000 microservices run on the cluster of 500,000 containers, tracing over 250 billions of RPC-based microservice calls per day.

Figure 6 shows the daily rates of real time microservice invocation amount and chain amount traced by JCallGraph in December 2018. All the transactions at JD (including JD Retails [10], JD Finance [8], JD Logistics [9] etc.) averagely trigger approximate 250 billions of microservice calls per day shown in the left. Using JCallGraph can effectively trace such massive scale microservice invocation relationship, around 500 millions of invocation chain amount per day as shown in the right.

4.2 Performance of JCallGraph

In this section, we evaluate the performance of JCallGraph from three key aspects through stress tests: the impact to applications caused by JCallGraph, the performance impact of various message size in log transfer layer, and the impact of various sampling rate.

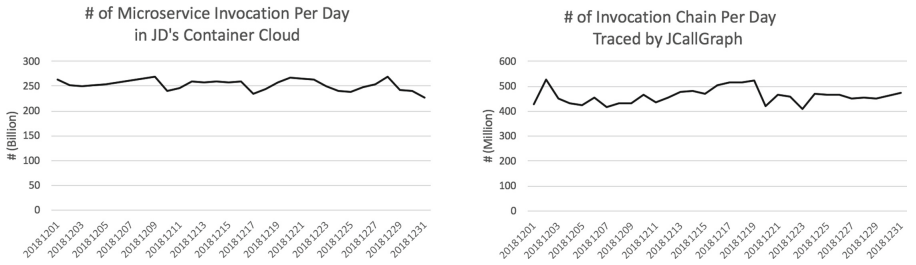

Fig. 6. One Month Operation Data in Dec. 2018. The transactions at JD.com averagely trigger approximate 250 billions of microservice calls per day shown in the left, JCallGraph records around 500 millions of invocation chain amount per day shown in the right.

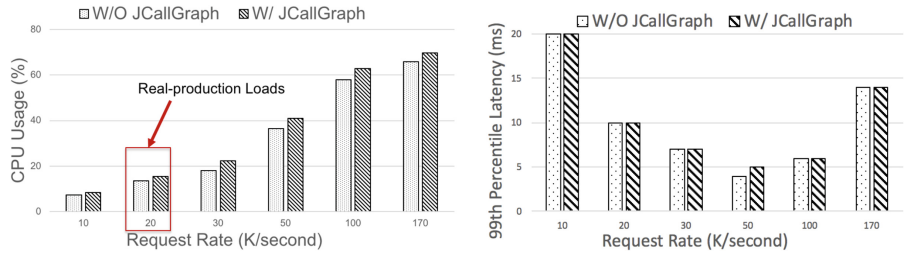

Fig. 7. Impact to applications under stress test when JCallGraph tracing is on and off. When the load of application is 20K per second, that is the usual load in our real-production environments, the around 1% overhead increase cause by JCallGraph has negligible impacts on applications. Meanwhile the 99th percentile latency of the applications in two scenarios is almost the same.

Impact to Applications Caused by JCallGraph. We first evaluate the impact to applications caused by JCallGraph via stress test in a test environment, where we deploy four client machines and one application server. Each machine is equipped with Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz, 32 GB memory and 1 GB network. We use HP LoadGenerator [7] to generate the workloads of stress test in the client machines. We generate various workloads, the maximum workload did not go beyond 170K because we observe that the load of applications does not exceed more than 30K most of the time in real-production environment. We deploy one application in server machines to serve the requests from the clients, the application invokes middlewares (e.g., JSF) meanwhile the tracing primitives of JCallGraph are invoked in the middlewares. We compare the overhead of the server and 99th percentile latency of the applications in two scenarios: switching on and off the tracing of JCallGraph.

Figure 7 illustrates the average CPU usage of the server and 99th percentile latency of the applications, when the number of requests handled by the server increases from 10K to 170K and JCallGraph tracing is on and off, usages of other resources (e.g., memory) are omitted since they are almost the same. We see that

as the load from client machines continues to increase, the CPU usage of the server with JCallGraph on and off increases correspondingly as the application needs to handle more requests. JCallGraph has a maximum of CPU overhead of 4.9% when the load arrives at 170K. Since the load of application in production is usually around 20K, a JCallGraph overhead of 1.9% has negligible and acceptable impacts on applications. The 99th percentile latency of the applications in the two cases is almost the same under various workloads.

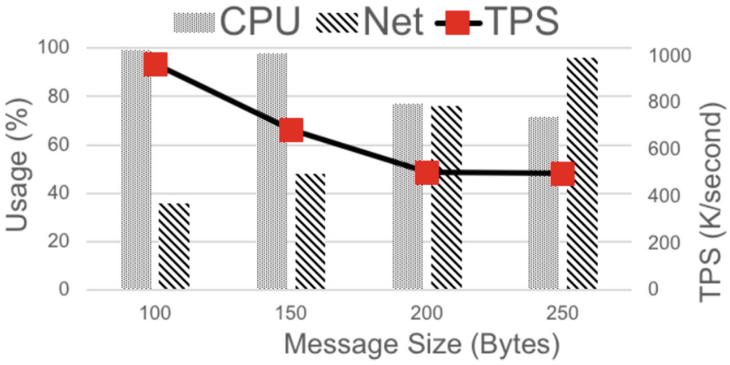

Fig. 8. The throughput and resource usage of log transfer layer with the various size of the message. It confirmed that the message size of 112 bytes, the actual message size in our real-production applications, can achieve the maximal throughput and resource usage environments.

Performance of Log Transfer Layer. We next study the performance of log transfer layer with the various size of the message carrying the information of traced microservice invocation relationships. In this experiment, we use one docker container as the transfer server and 50 docker containers as clients to send log messages. The server is equipped with 4 CPU cores, 8G memory, 50G disk and 10G ethernet.

Figure 8 illustrates how the length of messages impacts the throughput of log transfer layer and the resource usage. The y axis on the right refers to the maximum loads from clients which the server can sustain without losing any packages. We see that as the length of messages increases from 100 bytes to 250 bytes, the throughput of the transfer server decreases and the CPU usage drops from 99% to 72%. The reason is that as the messages size increases, the network bandwidth becomes the bottleneck in the real-production test environment leading to loss a lot of packages.

In real applications, the actual message size is around 112 bytes. Thus this result in Fig. 8 confirmed that the current context information and message size can achieve the maximal throughput and resource usage in our real-production environments.

Impact of Sampling Rate. We examine how the sampling rate reduces the number of packages sent per second, the network consumption as well as the burden of transfer server. In Fig. 9, we increase the client loads to report the number of packages sent by the application in log scale in various sampling rate from 1 to 4000. In all cases, we can successfully construct the normal invocation relationship. We see that the number of packages transferred without sampling compared with the number with sampling turned on is mostly proportional to the sampling rate. That means as the heavier the load the more network bandwidth consumption is saved.

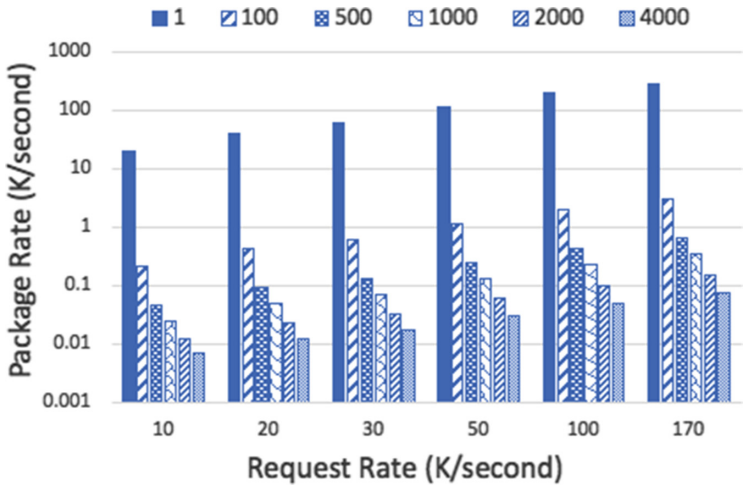

Fig. 9. Impact of sampling rate. Lower sampling rate can reduce the traffic pressure to the transfer layer.

JCallGraph uses different policy to record the invocation information for successful and failure microservice calling, and the accuracy of successful invocation context can be guaranteed by retry policy, thus we can use very low sampling rate to reduce the network pressure of transfer server.

4.3 Use Cases

In this section we share some use cases at JD.com, how JCallGraph helps developers to understand interactions among microservices, detect the problems and pinpoint the root-cause, analyse the dependencies of microservices.

Microservice Visualization. JCallGraph provides an interactive interface allowing users to visually explore the relationship across all microservices. Using the visualization, the developers can now then easily view the microservice invocation relationship. Figure 10 shows real-production application examples

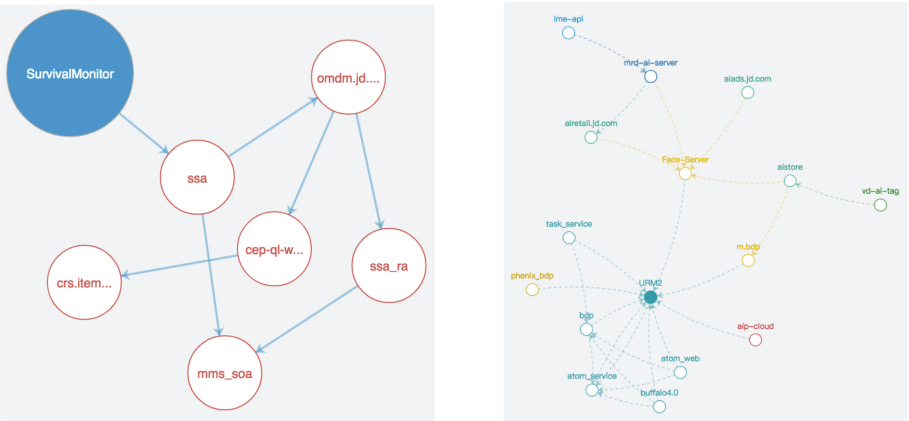

Fig. 10. Snapshots of microservice invocation graphs for real-production applications.

of microservice visualization. Intuitively, the developers free themselves from the complex and massive microservices, effective manage tens of thousands of microservices at JD.com. For example, JCallGraph helps to track the transition of products within the business flow of a Storm[2] application by tracing the processing flow of stock keeping unit (SKU)[3] of products.

Root Cause Analysis. Another important use cases of JCallGraph in JD.com is to help identify performance bottleneck and diagnose root causes [21–23,27]. JCallGraph provides an intuitive view to understand context allowing developer to quickly identify the location of errors with a meaning dynamic calling chain context. Figure 11 shows an example of invocation chain of an application where the exceptions occur and are marked as red. Note that the latency information is conveniently available showing that the top level exception takes around 50 ms. When an exception occurs, this call chain information can be easily accessed with the traceID that is injected into the application log. Based on this information, developers can quickly locate the corresponding portion of code and the runtime machine that produces the error and analyse why the error occurs under the current dynamic execution path.

Invocation Dependency Analysis. In addition to help identify root causes when exceptions or anomalies occur, JCallGraph offers various detailed analysis in terms of invocation dependencies and frequency analysis. We define two different types to represent the severity of dependency in JCallGraph, *strong dependency* and *weak dependency*. A call in an invocation chain has strong dependency when the call fails, all the subsequent calls can not be invoked, whereas a call

[2] Apache Storm. "http://storm.apache.org/".
[3] A stock keeping unit is a product or service identification code.

Microservice Name	Status	Latency
ShopGoodsService.getShopGoodsByParam	Success	7ms
SoService.checkReentrantSoMain	Success	5ms
WarehourseService.getWarehouse	Success	2ms
SoService.transportSo	Soft error	50ms
StockProcessServicd.occupySoStock	Soft error	5ms
DeptService.getDept	Success	1ms
SoService.addEcip2CloudTask	Success	8ms

Fig. 11. An example of root cause analysis. This figure shows errors of an invocation chain recorded by JCallGraph, we can further inspect the microservices with error and long latency to pinpoint the root cause.

L	Microservice Name	LR(%)	DR	CR	Note
0	Isv.EclpEdiOpenService .cancelSo	51.0	1.00	1.00	Strong dependency
1	Bbp.so.SoService .cancelSo	41.1	1.42	1.31	Strong dependency
2	Bbp.stock.StockProcess .Service.cancelSoStock	6.50	0.55	0.14	Strong dependency
1	Bbp.so.SoService .querySoMainByIsvSoNo	0.01	0.01	0.03	Weak dependency
1	Bbp.so.SoService .querySoMainByIsvSoNoPlus	0.01	0.01	0.04	Strong dependency
1	Bbp.so.SoService .addEclp2CloudTask	0.59	0.80	0.34	Weak dependency
1	Bbp.so.SoService .querySoMain	0.75	1.43	1.30	Strong dependency

Fig. 12. An example of invocation chain analysis. LR, DR, CR can be used to analyse the bottleneck, two types of frequency dependency as a caller and a callee. [L- level, LR - response time ratio, DR - dependency ratio, CR - calling ratio]

with weak dependency is the one when the call fails, some or all the subsequent calls are still invoked. Figure 12 marks strong and weak dependency in an invocation chain. Errors occur on calls with strong dependency are more severe than calls with weak dependency and usually have higher priority to be fixed.

Invocation frequency is another interesting insight for development and operation. We define two types of frequency dependencies in JCallGraph. One is the frequency as a callee in an invocation chain using the metrics of calling ratio

(CR), and the other is the frequency as a caller invoking other microservices in an invocation chain using the metrics of dependency ratio (DR). Figure 12 shows an example of invocation frequency analysis in an invocation chain. One successful use case of using invocation frequency analysis at JD.com is to plan the resource capacity. For example, JD.com launched its annual 6.18 Shopping Festival [1], we need to accurately estimate the usage frequency of microservices and plan the capacity and significantly improve the resource utilization while guaranteeing the performance of front-end on-line services during the period of annual 6.18 Shopping Festival.

Other Use Cases. There are other use cases of JCallGraph at JD.com such as entrance point and source analysis. Entrance point analysis identifies the first call in the calling chain where source analysis recognizes the callers and the callees of the current method. Entrance point analysis together with frequency analysis is used for capacity planning for shopping seasons or promotional sales days. As entrance points are the first call in the chain which can be invoked by frontend applications. It is easier to estimate the capacity from top to down following the chains. Source analysis is useful in that it helps developers to know exactly the upstream and downstream applications. Such information is used for new feature development and rolling updates.

5 Related Work

End-to-end request-flow tracing techniques are extensively implemented in distributed systems to support various performance monitoring tasks, such as Google's Dapper [24], Mace [11], Stardust [19], X-Trace [6], Retro [13], Pivot Tracing [14] etc. They gives us a lot of insights to design and implement JCallGraph to guarantee low overhead, such as sampling, trace points. However, they are not applied in such massive scale container environment.

OpVis [15] implements a monitoring and analytics framework in container environments to provide visualization and analytics, and DocMan [12] adopts the approaches such as distance identification and hierarchical clustering for detecting containerized application's dependencies, which are orthogonal and complementary to our works. JCallGraph, as the microservice monitoring platform at JD.com, provides a billion-scale invocation graph visualization as shown in Fig. 6 in Sect. 4.1, that is beyond the grasp of OpVis. More importantly, we can visualize the interactive microservices invocation graph in realtime, incur negligible overhead and not intrude applications, which can be practical in real-production industrial scenarios.

6 Conclusion

We presented JCallGraph, a tracing and analytics tool to visualize the microservice invocation graph for massive scale microservice platform in enterprise datacenter. Our evaluation and experiences at JD.com show that JCallGraph can

accurately capture the real-time invocation relationship among tens of thousands of microservices in millions of containers, one of the largest Kubernetes clusters in real production in the world, while achieving minimal overhead without any significant performance impact on real-production applications, and zero-intrusion to the code of applications. We hope that sharing our practice with the massive scale tracing system will provide insights to solve the challenges of monitoring and managing tens of thousands of microservices in enterprise data-center.

References

1. JD.com's 6.18 Shopping Festival. https://www.fungglobalretailtech.com/news/jd-coms-6-18-shopping-festival-just-discounts/
2. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 201–215. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33313-7_15
3. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
4. Bernstein, D.: Containers and cloud: from LXC to docker to kubernetes. IEEE Cloud Comput. **3**, 81–84 (2014)
5. Binder, W., Hulaas, J., Moret, P.: Advanced Java bytecode instrumentation. In: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, pp. 135–144. ACM (2007)
6. Fonseca, R., Freedman, M.J., Porter, G.: Experiences with tracing causality in networked services. INM/WREN **10**, 10 (2010)
7. HP: LoadRunner. https://www.claudihome.com/html/LR/WebHelp/Content/Controller/toc_MainController.htm
8. JD: Finance. https://jr.jd.com
9. JD: Logistics. https://www.jdwl.com
10. JD: Retails. https://www.jd.com
11. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: language support for building distributed systems. In: ACM SIGPLAN Notices, vol. 42, pp. 179–188. ACM (2007)
12. Liu, P., et al.: A toolset for detecting containerized application. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 194–201. IEEE (2018)
13. Mace, J., Bodik, P., Fonseca, R., Musuvathi, M.: Retro: targeted resource management in multi-tenant distributed systems. In: NSDI, pp. 589–603 (2015)
14. Mace, J., Roelke, R., Fonseca, R.: Pivot tracing: dynamic causal monitoring for distributed systems. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 378–393. ACM (2015)
15. Oliveira, F., Suneja, S., Nadgowda, S., Nagpurkar, P., Isci, C.: OpVis: extensible, cross-platform operational visibility and analytics for cloud. In: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track, pp. 43–49. ACM (2017)
16. Pahl, C.: Containerization and the PaaS cloud. IEEE Cloud Comput. **2**(3), 24–31 (2015)
17. RedisLabs: Redis. https://redis.io

18. Sambasivan, R.R., Shafer, I., Mace, J., Sigelman, B.H., Fonseca, R., Ganger, G.R.: Principled workflow-centric tracing of distributed systems. In: Proceedings of the Seventh ACM Symposium on Cloud Computing, pp. 401–414. ACM (2016)
19. Sambasivan, R.R., et al.: Diagnosing performance changes by comparing request flows. In: NSDI, 5, p. 1 (2011)
20. Sandoval, R., et al.: A case study in enabling DevOps using Docker. Ph.D. thesis (2015)
21. Shan, H., et al.: E-diagnosis: Unsupervised and real-time diagnosis of small-window long-tail latency in large-scale microservice platforms. In: Proceedings of the 2019 World Wide Web Conference on World Wide Web (2019)
22. Shan, H., Wang, Q., Pu, C.: Tail attacks on web applications. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1725–1739. ACM (2017)
23. Shan, H., Wang, Q., Yan, Q.: Very short intermittent DDoS attacks in an unsaturated system. In: Lin, X., Ghorbani, A., Ren, K., Zhu, S., Zhang, A. (eds.) SecureComm 2017. LNICST, vol. 238, pp. 45–66. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78813-5_3
24. Sigelman, B.H., et al.: Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc. (2010)
25. Thönes, J.: Microservices. IEEE softw. **32**(1), 116 (2015)
26. TIG: JDOS: a kubernetes-based datacenter operating system. https://github.com/tiglabs/jdos
27. Zhang, S., Shan, H., Wang, Q., Liu, J., Yan, Q., Wei, J.: Tail amplification in n-tier systems: a study of transient cross-resource contention attacks. In: ICDCS (2019)