Inferring Calling Relationship Based on External Observation for Microservice Architecture

Conference Paper · October 2017

DOI: 10.1007/978-3-319-69035-3_16

2 authors, including:

Shinya Kitajima
Fujitsu Ltd.
22 PUBLICATIONS **29** CITATIONS

SEE PROFILE

Inferring Calling Relationship Based on External Observation for Microservice Architecture

Shinya Kitajima and Naoki Matsuoka

Software Laboratory, Fujitsu Laboratories Limited, 4-1-1 Kamikodanaka, Nakahara, Kawasaki, Kanagawa 211–8588, Japan
{kitajima.shinya,matsuoka.naoki}@jp.fujitsu.com

Abstract. In recent years, a web service architecture namely microservices has attracted attention. Although the microservice architecture provides various advantages, it also has the disadvantage of making the root cause analysis of the complicated system. For root cause analysis, it is important to know the calling relationships between services since the service may call the other service and the latency of the called service may be wrong. Therefore, in this paper, we propose a method to infer the calling relationship between the services from communication logs observed from outside of the services.

Keywords: Microservice architecture, distributed tracing system, visualization, PaaS

1 Introduction

In recent years, a web service architecture namely microservices has attracted attention. In the microservice architecture, a web system is structured as a collection of loosely coupled services, and the services in the microservice architecture communicate via web APIs [5]. Microservice architecture was originally a spontaneously generated technique from companies with huge web system, such as Amazon and Netflix. Lewis *et al.* named the technique microservices [4].

There are various advantages of microservice architecture, including diversity of technologies, resilience, scalability, reliability, reusability and agility [2]. There are disadvantages, however, to constructing a web system as a distributed system [3], one of which is the complexity of root cause analysis. For root cause analysis, it is important to know the calling relationships between services since the service may call the other service and the latency of the called service may be wrong. It is thus difficult to discover the cause of a processing delay or error from the services since there are dozens or hundreds of services in a microservice architecture.

In the microservice architecture, a distributed tracing system, such as OpenZipkin[1] and OpenTracing[2], are the most famous methods for solving this problem. However, in order to use OpenZipkin and OpneTracing, it is necessary to

[1] http://zipkin.io/
[2] http://opentracing.io/

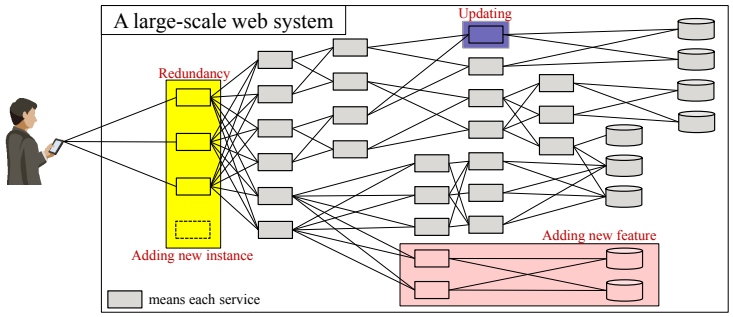


Fig. 1: The assumed environment.

modify the source code of each service. This forms large barriers toward the use of these methods since developers must add codes and libraries that are specific to the distributed tracing system. In addition, Aguilera *et al.* [1] proposed methods that can trace the system without modifications to the source code of the services, their methods do not consider reusability, scalability and agility of the microservice architecture.

Therefore, in this paper, we propose a method for inferring the calling relationship between the services from communication logs observed from outside of the services. We call the trigger message as the *parent message*, and the triggered message by the parent message as the *child message*.

The remainder of this paper is organized as follows. We describe our assumptions in Section 2. We explain our method for inferring the parents from communication logs in Section 3. We then evaluate the accuracy of our method in Section 4. Finally, we provide our conclusions in Section 5.

2 Assumptions

The typical examples of systems based on microservice architecture are A huge EC site and online movie site. Fig. 1 shows our assumed environment. Each service is developed and operated by the independent team. In addition, each service is constantly updated and new services are added to the system at any time. Each service runs in cooperation with loosely coupled with other services.

Each service is often executed as a container application, since the characteristics of the container, such as portability and agility, match well for the microservice architecture. A container management infrastructure, such as Kubernetes[3] and Cloud Foundry[4], is often used to manage the container.

The advantages of the microservice architecture are various, including diversity of technologies, resilience, scalability, reliability, reusability and agility. However, in the microservice architecture, it is very difficult to find out the

[3] https://kubernetes.io/
[4] https://www.cloudfoundry.org/

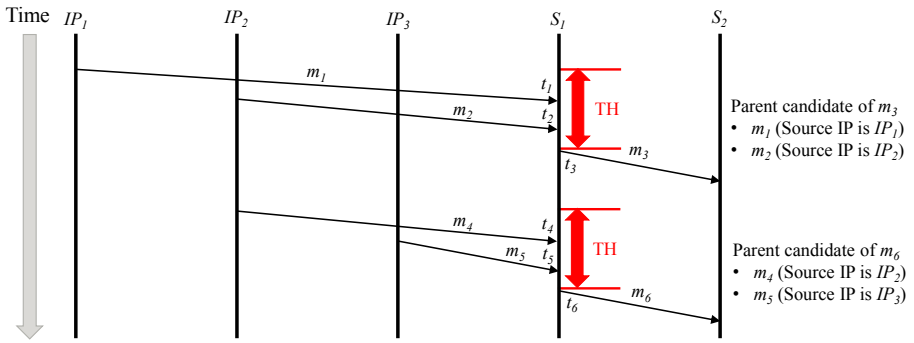


Fig. 2: Overview of proposed method for inferring the method.

cause when a processing delay or an error occurs, since the configuration of the system changes from time to time. Therefore, it is important to know the calling relationships between services in the microservice architecture. We call the visualized calling relationships between the services *flow*.

3 Method for inferring the parents

In this section, we propose a method for inferring the calling relationship between the services in the microservice architecture using communication logs (sender, receiver, and timestamp) that can be acquired on each server. Our method assumes that each message was triggered by another message, and collects communication logs of messages received within a given threshold from the sent time of the child message as the *parent candidate messages*. After our method has performed this process for a certain number of messages and obtained a set of parent candidate messages, it infers the parent messages heuristically.

3.1 Enumeration of the parent candidate messages

Our method infers the calling relationship by assuming that there is another message that triggered the message of interest. In general, web applications are listening for connections from clients on specific ports with specific IP addresses. Here, we define an IP address as IP and the pair of listening IP address and port of a service as S. In addition, we define a mapping function $getip : S \to IP$.

Fig. 2 shows that our method focuses on S_1 and enumerates all of the messages of which the source IP address is $getip(S_1)$ (m_3 and m_6 in the figure) and that are received within the threshold TH from the sent time of m_3 and m_6 (t_3 and t_6 in the figure) as the parent candidate messages (m_1 and m_2 for m_3, and m_4 and m_5 for m_6 in the figure). We cannot determine which message is the parent message if the service receives messages from multiple IP addresses within the threshold TH. We suppose that the optimal TH is different for each service. In this paper, for simplicity, we use the same TH for all services.

Table 1: From the communication logs acquired on the host H_x, the set of the parent candidate messages for communication sent to the service $s_{x,y}$

	Source IP addresses of parent candidate messages
ID1	IP_1
ID2	IP_1, IP_3
ID3	IP_1, IP_2, IP_4, IP_1
ID4	IP_2, IP_5
ID5	IP_1, IP_1, IP_4, IP_1
ID6	IP_2, IP_3
ID7	*None*

The amount of communication logs that our method uses for the inferring varies depending on the purpose. For example, we acquire the communication logs at all times, and when a failure occurs, our method uses the communication logs of the past hour to infer the flow to identify the cause.

We define the number of hosts as p and the set of communication logs acquired on a host $H_k(1 \leq k \leq p)$ as $M_k = \{m_{k,1}, m_{k,2}, \ldots, m_{k,n_k}\}$. Moreover, we define the source IP address of a communication $m_{k,i}$ as $IPs_{k,i}$, the destination IP address as $IPr_{k,i}$, the destination port number as $Pr_{k,i}$, and the occurrence time as $t_{k,i}$. $t_{k,i}$ is the sent time if H_k sent $m_{k,i}$, and $t_{k,i}$ is the received time if H_k received $m_{k,i}$. We define the set $CM_{k,i}$ of the parent candidate messages for $m_{k,i}$. $CM_{k,i}$ is the set of all communications for which the source IP address is the same as the source IP address $IPs_{k,i}$ of $m_{k,i}$, and for which the occurrence time $t_{k,j}$ is before $t_{k,i}$ and after $t_{k,i} - TH$. So, $CM_{k,i}$ can be represented as,

$$CM_{k,i} = \{m_{k,j} \mid IPr_{k,j} = IPs_{k,i}, t_{k,i} - TH < t_{k,j} < t_{k,i}\}(1 \leq j \leq n_k). \quad (1)$$

3.2 Aggregation of the parent candidate messages

In this section, we aggregate the enumerated set of parent candidate messages and calculate how often the parent candidates appear, among those appearing in message from a certain host to a certain service. First, among the communication logs acquired by a certain host H_k, we enumerate the parent candidates sent to the service $s_{k,i}(1 \leq i \leq ns_k)$. Table 1 shows an example of the enumerated parent candidates. Here, if two or more services are running on one host, it is necessary to extract only the communication logs by the specific service in the host and enumerate only the parent candidates sent to the service. The IDs in the table are allocated in order of occurrence of communication from host H_x to service $s_{x,y}$. As for ID3 and ID5, there are cases in which a parent candidate message includes an IP address more than once among its source IP addresses.

We count the number of the source IP addresses for each parent candidate in the set of parent candidate messages enumerated in the above. Table 2a shows the result of counting these using the information in Table 1. If there is no parent candidate message, we count the number as "*None*." Moreover, when a source

Table 2: Number of source IP addresses appearing as parent candidate messages.

(a) The communication logs acquired by host H_x.

	IP_1	IP_2	IP_3	IP_4	IP_5	*None*
ID1	1	0	0	0	0	0
ID2	1	0	1	0	0	0
ID3	1	1	0	1	0	0
ID4	0	1	0	0	1	0
ID5	1	0	0	1	0	0
ID6	0	1	1	0	0	0
ID7	0	0	0	0	0	1
Total	4	3	2	2	1	1

(b) The result excluding candidates having IP_1.

	IP_2	IP_3	IP_4	IP_5	*None*
ID4	1	0	0	1	0
ID6	1	1	0	0	0
ID7	0	0	0	0	1
Total	2	1	0	1	1

IP address appears more than once for a given parent candidate message, we only count it once. This is because several communications that have the same source IP address are included in a set of parent candidate messages when the access frequency from the client is high.

3.3 Inferring the parent message

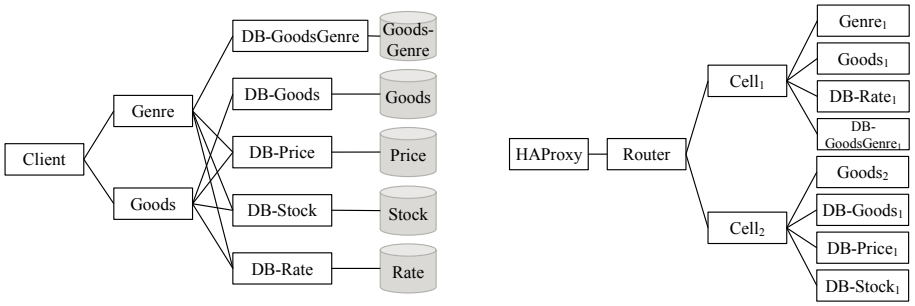
In this section, we guess the parent based on the result of Section 3.2 with the following steps.

1. We select the parent candidate massage for which the source IP address appears most frequently as the parent message.
2. We check whether the parent message is included in each set of parent candidate messages, and exclude those set of parent candidate messages that include the parent message.
3. We go back to the Step 1 while a set of parent candidate messages exists.

As an example, we apply these steps to Table 2a. In Step 1, IP_1 is selected as the parent message since it has the highest total number of appearances. Next by applying Step 2, we get Table 2b by excluding the sets of the parent candidate messages that include IP_1. After that, we apply Step 3, which takes us back to Step 1 by which we select IP_2 as the parent message, since it has the most appearances at this point. In this way, we apply these steps until we finally select "None" as the parent message and conclude the process. As a result, we select the communication from IP_1 to H_1, IP_2 to H_1, and "None" as the parent message of the communication from H_x to $s_{x,y}$.

4 Evaluation

In this section, we evaluate our method according to the following three criteria. We define the set of parent messages in the inferred result as C_{result}, the set of correct parent messages, which we defined in advance, as $C_{correct}$, and the intersection of these sets as $C_{TP} = C_{result} \cap C_{correct}$.



(a) Configuration diagram (b) Configuration on Cloud Foundry

Fig. 3: Configurations.

- **Precision:** The ratio of the number of messages included in the set the correct parent messages divided by the number of parent messages included in the inferred result ($Precision = |C_{TP}|/|C_{result}|$).
- **Recall:** The ratio of the number of messages included in the set of parent messages in the inferred result divided by the number of correct parent messages ($Recall = |C_{TP}|/|C_{correct}|$).
- **F-measure:** Defined as follows using Precision and Recall:

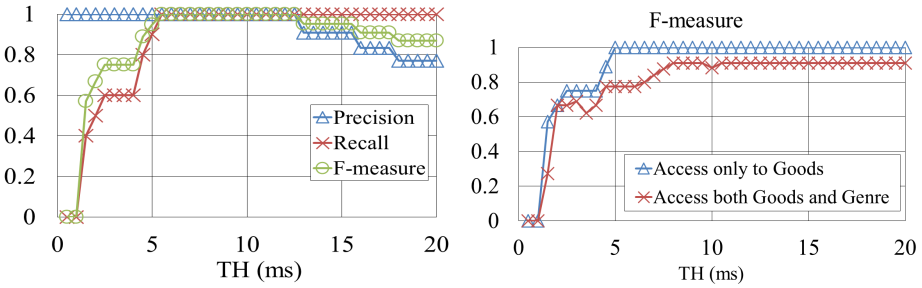
$$F = \frac{2 \times Precision \times Recall}{Precision + Recall}. \tag{2}$$

4.1 Evaluation environment

For the evaluation, we used a system based on microservice architecture that we implemented by imitating an EC site. Fig. 3a shows the configuration of this system. This system consists of seven services, five of which are services connected to the database and the other two of which are services accessing multiple services. The Genre service and the Goods service each call four services that connect to the database at the same time. After all results are returned to them, they return results to the user. The implementation languages of the services and the databases are Go and MySQL, respectively.

Each service is running as an application on Cloud Foundry. A unique URL is allocated to each application, and we can access each application using that URL. Fig. 3b shows the communication path to the applications inside Cloud Foundry. The access from the client to the application is sent to the Router via HAProxy, and the Router allocates access to instances of each application. When there are multiple instances of an application, access is sequentially allocated in a round-robin manner. Each application instance is running as an application container on the Cell, and each application listens for access at a specific port on the Cell.

If we applied the method as it is to the Cloud Foundry environment, the parent message of all messages would be message from the HAProxy to the



(a) Precision, Recall, and F-measure when changing TH.

(b) Effect of changing the number of services on the F-measure.

Fig. 4: Evaluation results.

Router. Therefore, we added a process to check which message calls the message from the HAProxy to the Router and to rewrite the sender of the message. That is, the source of the message from the HAProxy to the Router is replaced by client or one of the services.

We define the message from component A to component B as $[A, B]$. In addition, we call the Goods service and the Genre service *Front* because client accesses them first, and we call the other services, which access the database, *Backend*.

4.2 Evaluation result

Impact of TH Fig. 4a shows the evaluation result of Precision, Recall and F-measure of our method while changing TH from 0.5 to 20 ms. In this evaluation, clients only access the Goods service, and a client's access pattern is equal access once every 5 s for 5 min. We set the number of each service to 1.

The results show that Recall is low when TH is low, and Precision is low when TH is high. As a result, the F-measure is highest when $5.5 \leq TH \leq 12.5$. When TH is low, there are few messages included in the set of parent candidate messages, and the correct parent message is not included in the result. On the other hand, when the TH is high, the set of parent candidate messages includes message that is not correct parent message, and the result also includes message that is not correct.

The breakdown of the incorrect answers shows that $[Client, Router]$ is selected as the parent message of $[Router, Backend]$ in addition to $[Goods, Router]$, which is the correct answer, resulting in low Precision. In the system that we used for the evaluation, first the client accesses the Goods service, and then the Goods service calls the Backend. Therefore, when TH is high, $[Client, Router]$ is included as parent candidate message of $[Router, Backend]$, resulting in low Precision.

Impact of the number of services Fig. 4b shows the F-measure when the client accesses only the Goods service and when the client accesses both the

Goods and the Genre services. We set the number of instances of each service to 1, and we vary TH from 0.5 to 20 ms. In this evaluation, the client's access pattern is equal access once every 5 s. When the client accesses both the Goods and Genre services, the client's access pattern is equal access once every 5 s for both the Goods and Genre services, and the Goods and Genre services are accessed at almost the same time every time.

The results show that the F-measure is lower when the client accesses both the Goods and Genre services. When the client accesses both the Goods and Genre services every time, our method cannot determine whether $[Goods, Router]$ or $[Genre, Router]$ is the correct parent message of $[Router, Backend]$. This is because $[Goods, Router]$ and $[Genre, Router]$ are included in the set of parent candidate messages almost every time, and our method selects either $[Goods, Router]$ or $[Genre, Router]$ as the parent.

In a real situation, we suppose that it is unrealistic for a client to access the same several services at almost the same time every time. Even in such a situation, the F-measure is more than 0.9 when $TH > 10$, and our method can mostly infer the calling relationship correctly.

5 Conclusion

In this paper, we proposed a method to infer the calling relationship between the services from the communication logs observed outside the services, such as sender, receiver, and timestamp. Our method assumes that each message is triggered by another message, and infers the trigger message considering scalability and agility of the microservice architecture. The evaluation results show that our method can infer the calling relationships with high accuracy.

In the future, we will propose a method that automatically defines the optimal TH. In addition, we will confirm and improve the performance of our method in real systems based on microservice architecture.

References

1. Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P., Muthitacharoen, A.: Performance debugging for distributed systems of black boxes. In: 9th ACM Symposium on Operating Systems Principles (SOSP), pp. 74–89 (2003)
2. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables DevOps: migration to a cloud-native architecture. In: IEEE Software, vol. 33, No. 3, pp. 42–52 (2016)
3. Ciuffoletti, A.: Automated deployment of a microservice-based monitoring infrastructure. In: Procedia Computer Science, vol. 68, pp. 163–172 (2015)
4. Lewis, J., Fowler, M.: Microservices. `http://martinfowler.com/articles/microservices.html` (2014)
5. Newman, S.: Building microservices, designing fine-grained systems. O'Reilly Media (2015)