Using Vector Clocks to Monitor Dependencies among Services at Runtime

Daniele Romano
Software Engineering Research Group
Delft University of Technology
The Netherlands
daniele.romano@tudelft.nl

Martin Pinzger
Software Engineering Research Group
Delft University of Technology
The Netherlands
m.pinzger@tudelft.nl

ABSTRACT

Service-Oriented Architecture (*SOA*) enable organizations to react to requirement changes in an agile manner and to foster the reuse of existing services. However, the dynamic nature of Service-Oriented Systems and their agility bear the challenge of properly understanding such systems. In particular, understanding the dependencies among services is a non trivial task, especially if service-oriented systems are distributed over several hosts and/or using different SOA technologies.

In this paper, we propose an approach to monitor dynamic dependencies among services. The approach is based on the vector clocks, originally conceived and used to order events in a distributed environment. We use the vector clocks to order service executions and to infer causal dependencies among services. In our future work we plan to use this information to study change and failure impact analysis in service-oriented systems.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*reverse engineering*; D.2.11 [**Software Engineering**]: Software Architectures—*Service-oriented architecture (SOA)*

General Terms

Management

Keywords

SOA, monitoring, dynamic dependencies

1. INTRODUCTION

IT organizations need to maintain strategic advantages in businesses in oder to stay competitive on the market. The main means to achieve such a goal is the ability to deploy updated or new applications before their competitors. In the IT world this capability is well known as *agility*. As a consequence of this need IT organizations started to conceive their software systems as Software as a Service *SaaS*, overcoming the poor inclination of monolithically architected applications towards *agility*. Hence, the adoption of Service Oriented Architecture (*SOA*) has become more and more popular. Besides *agility*, SOA-based application development also contributes to reduce development costs through service reuse.

On the other hand, mining dependencies in a flexible architecture such as a *SOA* has become increasingly relevant to understand the entire system and its evolution over time. The distributed and dynamic nature of those architectures make this task particularly challenging. Moreover, mining dependencies is even more expensive when dealing with architectures, like *SOAs*, built up of connecting legacy systems. In such a scenario, having updated documentations might be not always possible nor cost effective. Hence, a robust approach is necessary to infer dynamic dependencies among services.

To the best of our knowledge, the existing technologies used to deploy a Service-Oriented system do not provide such functionalities. There is still a lack of approaches to detect the execution traces and, hence, the entire chain of dependencies among services. For instance, open source Enterprise Service Bus systems (*e.g.*, MuleESB[1] and ServiceMix[2]) are limited to detect only direct dependencies (*i.e.*, invocation between pair of services). Such monitoring facilities are widely implemented through the *wire tap* and the *message store* patterns described by Hohpe *et al.* [5]. Other tools, like HP OpenView SOA Manager[3], allow the exploration of the dependencies, but they must explicitly be specified[1].

In this paper, we propose an approach based on vector clocks to monitor the dependencies among services in a *SOA* at run-time. The vector clocks have been originally conceived and used to order events in a distributed environment. We use them to order service executions and to infer causal dependencies. A dynamic analysis is necessary to reach a better accuracy. In fact, the existing static approaches are inadequate when applied to *SOAs*, due to their dynamic nature (*i.e.*, the dynamic binding feature). Moreover, a dynamic approach is useful also to monitor a system for debugging purposes.

QASBA '11, September 14, 2011, Lugano, Switzerland

[1] `http://www.mulesoft.org/`
[2] `http://servicemix.apache.org/`
[3] `http://h20229.www2.hp.com/products/soa/`

1.1 A motivating example

To highlight the relevance of this problem, consider the simple application shown in Figure 1, where the services are deployed in a distributed environment with two different hosts.

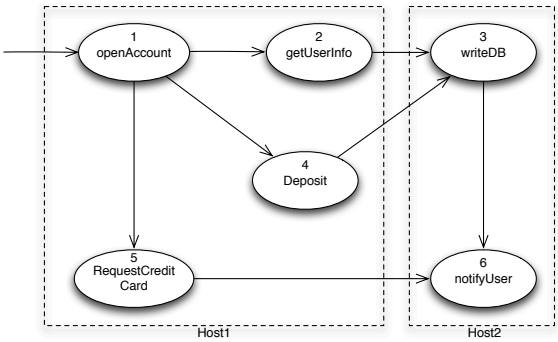

Figure 1: A simple service oriented system

Mining dependencies among services deployed on the same machine (*e.g.*, between services 3 and 6) might be straightforward through the facilities provided by a generic application server. Detecting that the invocation from the Service 3 to the Service 6 is triggered by the invocation from the Service 1 to the Service 2 can be a tricky problem [1]. Moreover, imagine the Service 6 is triggered only when the Service 3 is invoked by the service 2. Mining these dynamic dependencies can help to build execution traces and, hence, to obtain a better understanding of the service-oriented system.

This paper is structured as follows. In Section 2 we provide the background information on vector clocks and our reference *SOA*. In Section 3 we introduce our approach, and in Section 4 we present a possible implementation of it. Related work and conclusions are discussed in Section 5 and Section 6.

2. BACKGROUND

Even though SOA has become a reference model to design distributed systems, the different terminologies used in the modern world of IT can lead to significant misunderstandings. In this paper we use the term SOA to refer to an architectural model (1) that positions services as the primary means through which the solution logic is represented, and (2) that is neutral to any technology platform [3]. In this landscape, a *service* is a unit of solution logic that can be built and implemented as a component, a web service or a REST service.

In order to be as technology independent as possible, in the remainder of the paper we refer to a generic SOA as composed by an Enterprise Service Bus (*ESB*), as shown in Figure 2. In this paper, the *ESB* is referred to as an architectural pattern, which is an important part of a SOA, and not as a software product. We consider an *ESB* as a logical bus which eases the communication between services of an application. We leave out details about its implementation (*e.g.*, through a specific ESB software product or a connection of multiple ESBs).

Ordering events in a distributed system, such as a Service-Oriented System, is a challenging problem since the physical

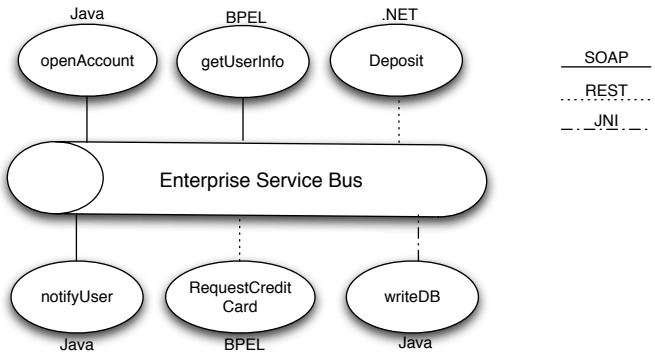

Figure 2: Enterprise Service Bus

clock of the different hosts may not be perfectly synchronized. The logical clocks were introduced to deal with this problem. Their original purpose was to give a consistent temporal ordering of events, rather than for determining cause and effect relationships. The first algorithm relying on logical clocks was proposed by Lamport [7]. This algorithm is used to provide a partial ordering of events, where the term *partial* reflects the fact that not every pair of events needs to be related. Lamport formulated the *happens-before* relation as a binary relation over a set of events which is reflexive, antisymmetric and transitive. According to his work, the notation $a \rightarrow b$ is used to express that the event a happens before the event b.

Lamport's work is a starting point for the more advanced vector clocks defined by Fidge and Mattern in 1988 [4, 8]. Like the logical clocks, they have been widely used for generating a partial ordering of events in a distributed system. Given a system composed by N processes, a vector clock is defined as a vector of N logical clocks, where the i^{th} clock is associated to the i^{th} process. Initially all the clocks are set to zero. Every time a process sends a message, it increments its own logical clock, and it attaches the vector clock to the message. When a process receives a message, first it increments its own logical clock and then it updates the entire vector clock. The updating is achieved by setting the value of each logical clock in the vector to the maximum of the current value and the values contained by the vector received with the message.

3. APPROACH

In our approach we use the vector clocks to order service invocations. Giving an order to the invocations allows to discover the execution traces and infer the dynamic dependencies among services. We conceive a vector clock (VC) as a vector/array of pairs *(s,n)*, where s is the service's id and n is number of times the service s is invoked. When the *ESB* receives an execution request for the service s, the vector clock is updated according to the following rules:

- if the request contains a *null* vector clock (*e.g.*, a request from outside the system), the vector clock is created, and the pair *(s,1)* is added to it;
- otherwise, if a pair with service's id s is already contained in the vector clock, the value of n is incremented by one; if not, the pair *(s,1)* is added to the vector.

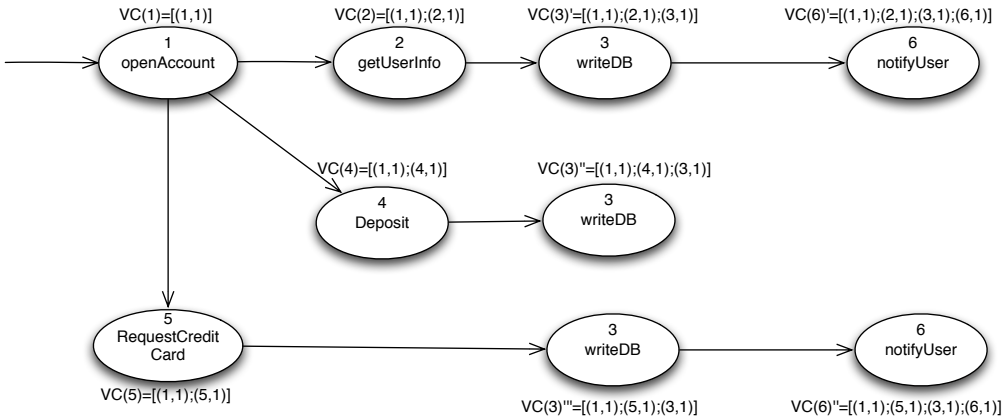

Figure 3: Example of the execution flow and vector clocks when the service *openAccount* is invoked

Once the vector clock is updated, its value is associated to the execution of service s and we label it *VC(s)*. The vector clock is attached to the messages sent to the other invoked services, hence, it is propagated along the execution flow.

With the set of vector clocks, we can infer the partial causal ordering of the service executions in a distributed Service-Oriented System. Given the vector clocks associated to execution of the service i and the service j, *VC(i)* and *VC(j)*, we can state that the service i depends on the service j, if *VC(i)* < *VC(j)*, according to the following equation:

$$VC(i) < VC(j) \Leftrightarrow \forall x\,[VC(i)_x \leq VC(j)_x] \wedge \exists x'\,[VC(i)_{x'} < VC(j)_{x'}] \quad (1)$$

where $VC(i)_x$ denotes the value of n in the pair *(x,n)* of the vector clock *VC(i)*. In other words, the execution of a service i causes the execution of a service j, if and only if all the pairs contained in the vector *VC(i)* have an n value less or equal to the corresponding n value in *VC(j)*, and at least one n value is smaller. If all the n values are equal except one value which is smaller, than we state that there is direct causality from service i to service j. Whether a pair *(s,n)* is missing in the vector the value n is considered equals to 0. To infer the execution traces, and hence the dynamic dependencies among services, we need to apply the binary relation in (1) among each pair of vector clocks.

Consider our example system from Figure 1, where vector clocks with superscripts mark vector clocks associated to different executions of the same service. The execution flow caused by the invocation of the service *openAccount* is shown in Figure 3, along with the vector clocks associated to each execution event. When the *openAccount* service is invoked, there is no vector clock attached to the message, since the invocation request comes from outside. Hence a new vector clock (*VC(1)*) is created with the only pair (1,1). Then the execution of the service *openAccount* triggers the execution of the service *getUserInfo*. When this service is invoked, a new pair (2,1) is added to vector clock, obtaining the new clock *VC(2)=[(1,1),(2,1)]*. When the service *Deposit* is invoked its vector clock is set to *VC(4)=[(1,1),(4,1)]*.

Consider the execution of the service *writeDB*, and imagine we want to infer all the services which depend on it. Since we have multiple invocations of the service *writeDB* in the execution flow, the dependent services are all the services x whose vector clock *VC(x)* satisfy the following boolean expression:

$$VC(x) < VC(3)' \vee VC(x) < VC(3)'' \vee VC(x) < VC(3)'''$$

These services are *openAccount*, *getUserInfo*, *Deposit* and *RequestCreditCard*.

If we want to infer all the services that *writeDB* depends on, we look for all the services x whose vector clock *VC(x)* satisfy the following boolean expression:

$$VC(x) > VC(3)' \vee VC(x) > VC(3)'' \vee VC(x) > VC(3)'''$$

The sole service is *notifyUser*.

4. IMPLEMENTATION

A generic *ESB* provides a logical bus to assist the services of an application communicate with each other. This is achieved through the *message broker* architectural pattern [5], which decouples the communicating endpoints and maintains control over the flow of messages. The *message broker* eases the implementation of functionalities such as routing, transformation, and connectivity. In order to integrate our approach into an *ESB* we can take advantage of this pattern.

Consider the simplified message flow shown in Figure 4. The *inbound* and *outbound transformer* are used to transform inbound and outbound data if they are not in the correct format. The *service invocation* is used to invoke the services. These 3 modules can easily be instrumented to implement our approach, as illustrated in Figure 4. First, in the *inbound transformer* we can implement the logic necessary to insert the vector clock if it is not already present in the message (*e.g.*, when the service is invoked from a source outside the system). Then, when the service is invoked, the vector clocks are updated according to the rule presented before. Finally, the *outbound transformer* is responsible to attach the updated vector clock to the outgoing message.

5. APPLICATIONS

The approach presented in this paper is the first attempt to assess the quality of Service Oriented Architectures. In

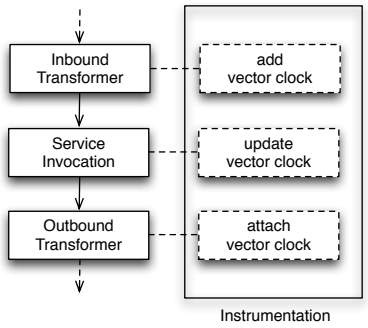

Figure 4: Simplified message flow in a *message broker* pattern and possible instrumentation to implement the approach

our experience we have found this task particularly challenging because of the lack of documentation. This is mainly due to the dynamic nature of the Service-Oriented systems (*e.g.*, dynamic binding). In fact, it is complex and difficult to analyze and understand Service-Oriented systems because the documentation does not exist or is incomplete. In this scenario, our approach can help to obtain a better understanding of the system through a complete dependency graph captured at runtime. This graph will help us to investigate the interaction between services and, hence, to define usage patterns and antipatterns that can affect the quality of *SOAs*. For instance, we plan to use our approach to measure various coupling and cohesion metrics for assessing the quality of the composition and orchestration of Service-Oriented systems.

6. RELATED WORK

The most recent work on mining dynamic dependencies has been developed by Basu *et al.* [1] in 2008. Basu *et al.* infer the causal dependencies through three dependencies identification algorithm, respectively based on the analysis of 1) occurrence frequency of logged message pairs, 2) distribution of service execution time and 3) histogram of execution time differences. However, the approach has been validated with simple scenarios. In 2004 Briand *et al.* [2] proposed a methodology and an instrumentation infrastructure aimed at reverse engineering of UML sequence diagrams from dynamic analysis of distributed java systems. Their approach is based on a complete instrumentation of the systems under analysis that requires a complete knowledge of the systems.

Winkler *et al.* [9] in 2009 proposed an approach to analyze dependencies between services in a composition. Their approach consists in detecting the dependencies at design time and it is aimed at validate the negotiate SLAs (service level agreements).

Finally it is necessary to cite the work developed by Hrischuk *et al.* [6], who provided a series of requirements to reverse engineer scenarios from even traces in a distributed system. However, besides the requirements, this work does not provide any approach to monitor dependencies in a Service-Oriented system.

7. CONCLUSIONS

In this paper, we have presented a novel approach to monitor dynamic dependencies among services using vector-clocks. They allow to reconstruct a more complete dependency graph from a service-oriented system at run-time. Such information is of great interest for both, researchers and developers of service-oriented systems.

For instance, researchers can benefit from our approach by using the information about dependencies to study service usage patterns and anti-patterns more accurately. In addition, researchers and developers can also use the information to identify the potential consequences of a change or a failure in a service, also known in literature as *change* and *failure impact analysis*. Moreover, the possibility to monitor the causal dependencies between services can be a useful tool for debugging service-oriented systems.

As future work, we plan to implement and validate the approach comparing the capability with the existing dependency monitoring facility (*e.g.*, the approach proposed by Basu *et al.* [1]) and in presence of complex scenarios. With this position paper, we are looking for constructive feedback that can help us to further improve our approach.

8. REFERENCES

[1] S. Basu, F. Casati, and F. Daniel. Toward web service dependency discovery for soa management. In *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*, pages 422–429, Washington, DC, USA, 2008. IEEE Computer Society.

[2] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Trans. Softw. Eng.*, 32:642–663, September 2006.

[3] T. Erl. *Soa: principles of service design*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2007.

[4] C. J. Fidge. Timestamps in message-passing systems that preserve partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[5] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[6] C. E. Hrischuk and C. M. Woodside. Logical clock requirements for reverse engineering scenarios from a distributed system. *IEEE Trans. Softw. Eng.*, 28:321–339, April 2002.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[8] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[9] M. Winkler, T. Springer, E. D. Trigos, and A. Schill. Analysing dependencies in service compositions. In *Proceedings of the 2009 international conference on Service-oriented computing*, ICSOC/ServiceWave'09, pages 123–133, 2009.