A Framework-based Runtime Monitoring Approach for Service-Oriented Software Systems

Cuiting Chen
Delft University of Technology
The Netherlands
cuiting.chen@tudelft.nl

Andy Zaidman
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

Hans-Gerhard Gross
Delft University of Technology
The Netherlands
h.g.gross@tudelft.nl

ABSTRACT

The highly dynamic and loosely coupled nature of a service-oriented software system leads to the challenge of understanding it. In order to obtain insight into the runtime topology of a SOA system, we propose a framework-based runtime monitoring approach to trace the service interactions during execution. The approach can be transparently applied to all web services built on the framework and reuses parts of information and functionality already available in the framework to achieve our goals.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*reverse engineering*

General Terms

Management

Keywords

runtime monitoring, SOA

1. INTRODUCTION

Today, many organizations deploy services for realizing their IT landscapes. They aim at exploiting the ability of the service technologies to integrate existing legacy components, and to better cope with changing business requirements. These are two core demands of industry which are addressed adequately through highly dynamic and loosely coupled service-oriented architectures (SOA) [14]. In particular, services can be discovered, bound and executed during operation time, enabling (online) evolution [8].

However, loose coupling and the highly dynamic nature of service-based software systems also present challenges in the maintenance and evolution processes. For example, the actual configuration of a system realized with services, and the usage of its parts, can only be seen at runtime [2]. Although online maintenance and evolution is technically well supported, system comprehension, a key prerequisite for conducting maintenance and evolution [16], is not [8]. Understanding complex SOA in order to plan and implement maintenance and evolution, is still one of the major challenges for software engineers [13].

Information that can be derived statically is not enough for understanding and visualizing how a SOA is deployed at runtime, and how the services interact in order to realize the business goals of various users. Instead, or in addition, runtime monitoring should be employed as the primary means to obtain data on the dynamic behavior of a SOA and its usage. In this way, software engineers can get a better understanding of the service-based software system and, consequently, they can plan and perform necessary system maintenance and evolution activities more adequately and timely. By also adding the usage information of individual services to the extracted views, the engineers can better plan maintenance, thus reducing the disturbances to the nominal system operation of an entire IT infrastructure. Moreover, online monitoring can facilitate SOA governance through supporting load balancing, identifying performance bottlenecks, or usage profiling.

In this paper, it is our goal to support software engineers by creating high-level views of how services (dynamically) interact, i.e., the *runtime topology*. In order to realize this, we first identify the associated monitoring data, that will subsequently help engineers in the (online) maintenance and evolution of service oriented architectures. Furthermore, in order to acquire the required information, we propose to extend existing service frameworks to support monitoring, and to be able to exploit information readily available inside these frameworks. For example, the addressing information used to send and receive requests and responses can be extracted to reversely reason about the invocation sequences and activities of users. That way, engineers can update and maintain parts of the SOA with low current usage, or they can defer maintenance to periods with expected low usage, thereby minimizing disturbance. Some of the data required for planning and realizing such maintenance activities are already provided by SOA frameworks through logging mechanisms contained in many platforms. Moreover, additional mechanisms can be integrated into frameworks, in order to provide required data according to various comprehension goals. For example, a framework can be extended to add a *sequence id* to SOAP messages, which provides the order of messages caused by an invocation traversing different machines. At present, our investigation is based on Apache

QASBA '11 September 14, 2011, Lugano, Switzerland

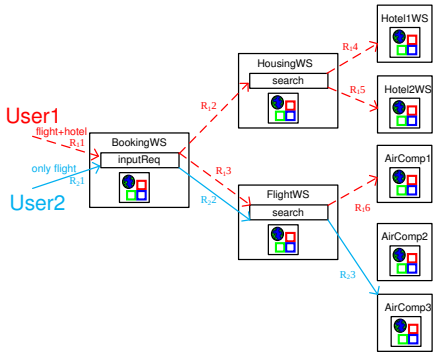

Figure 1: SOA system Scenario

Axis2/Java[1], which is an open-source web service framework implemented in Java and which provides various features to ease the development of web services.

In order to being able to comprehend a SOA, we formulate the following research questions:

RQ1 Which information is required to understand a SOA system?

RQ2 How can we exploit the information from a running SOA system?

RQ2.1 If the web service is built on a SOA framework, which information is already inside the framework?

This paper is structured as follows: in Section 2, we present the proposed approach for runtime monitoring, while Section 3 discusses our prototype-implementation based on the Axis2 framework. In Section 4, we address the aforementioned research questions and discuss the feasibility of the approach. Section 5 is about related work; and in Section 6 we conclude our paper with future work.

2. PROPOSED APPROACH

Goal.

Our basic goal is to support the comprehension process of complex SOA systems. In particular, we are interested in understanding the runtime topology of services, which entails obtaining insight into how services work together to execute a particular functionality. Recovering this type of information calls for a dynamic analysis approach, which means monitoring the SOA system during runtime. An additional benefit of adopting a dynamic analysis approach is that we are able to follow an *on-demand* comprehension strategy, i.e., we only deal with information relevant to the execution scenario and to the part that we want to understand [6]. In order to accomplish this dynamic analysis, we aim to integrate monitoring techniques into web service frameworks, as to leverage all available information inside the framework for monitoring. This approach (1) can be transparently applied to all web services built on the framework, and (2) parts of information and functionality already existing in the framework can be reused to achieve our goals.

Figure 1 presents the runtime scenario of a service-based system: customers with different inputs invoke a different set of services involved. In particular, you will see that depending on whether a customer requires a flight or a flight/hotel combination, a different set of services is invoked. In order to reconstruct the runtime topology of a SOA system, i.e., how services interact at runtime, we at least need to obtain the following data from the web service framework:

service id: before we deduce the interactions between services, we should first be able to identify the services involved. The service name is not enough, as there may be different services sharing the same name. Moreover, those services can be described in the same service description file with different *target namespaces*. Therefore, in order to uniquely identify a service, we propose a simple scheme using the combination of the URI of the service description file, the target namespace and the service name as the service id.

interface id: to further know which function a user is invoking in a service, it is also necessary to log the name of the invoked service interface. However, a service can contain two operations, i.e., interfaces, with the same name and different parameters. For example, earlier versions of WSDL[2], one of the most common web service description languages, support operation overloading. Hence, the information of parameters is also required to distinguish an operation.

process id: in order to trace an invocation passing through a number of web services, a specific identifier named *process id* is needed to link all requests and responses involved.

sequence id: as service-based systems are frequently deployed in a distributed context, using only time stamps to deduce the invocation sequence of the services might be problematic, since physical clocks in various machines may have different deviations. This problem can be mediated by using either a *logical clock* [10] or *vector clock* [7, 12], or by a simple mechanism, which involves adding a *sequence id* to the message that is being sent to the next service. For each request that comes into the SOA system, a new sequence id counter is created and each time this request causes a new message to be sent to another service, the sequence id is incremented.

SOA frameworks.

Generally, a web service consists of three major parts: a listener, a proxy, and the service implementation [15]. When a web service is created based on a service framework, typically the service developers only need to implement the core business logic in the service implementation, and the other two parts are realized in the framework. It is the listener's task to detect incoming and outgoing messages passing through the server. The proxy deals with messaging and addressing.

Once the listener receives an incoming request, it will forward the message to the proxy, which will parse the request to obtain the information of the invoking service and interface. Then the content of the request is delivered to the target service. After the service sends back the response, the proxy will decode the response and the listener will dispatch the result to the invoker. In order to execute the invocation properly, the service framework stores the addressing information to dispatch requests and responses at runtime. In addition, each service creates a specific instance of itself for each invocation, and an object id is assigned to the instance for the aim of identification.

[1] `http://axis.apache.org/axis2/java/core/`

[2] `http://www.w3.org/TR/wsdl`

Hence, some information required to rebuild the runtime topology of the system, such as the information for the service id and the interface id, is already inside the framework. However, obtaining the other information elements from the framework requires extra work. Generally, a framework does not offer a process id for each message. Thus, we can either extend the framework to enable the new id generation, or reuse the existing ids inside the framework. For example, it is feasible to reuse the object id of the service firstly invoked in a sequence as a process id (the mechanism to determine the first invoked service will be considered in future work). The framework keeps passing the id to all following messages involved in the same activity. After logging the information, we can identify all messages containing the same process id as belonging to the same invocation. For the sequence id, however, a particular mechanism should be added into the framework to guarantee its delivery and incrementation.

Service frameworks typically have a logging system in place to track abnormal behavior that might arise. It is our aim to reuse these monitoring mechanisms and extend them for our purposes.

3. PROTOTYPE IMPLEMENTATION

In order to operationalize our idea, we decided to use the Apache Axis2 framework (Java implementation) as our research vehicle. Axis2 is an open-source web service framework which supports various protocols for web services, such as SOAP, REST, etc. It also offers many features to ease the development of web services. Inside the framework, a set of *transport* components are built to send and receive messages, a core engine invokes a number of pre-required *handlers* to deal with part of the information in the message, and a handler named *message receiver* further delivers the message to the invoked service implementation for execution[3].

Axis2 maintains an *information model* to store all data inside the framework in two major categories: *Description* and *Context*. The static information, like deployment details of services, is kept inside the description. While all data related to the runtime execution are hierarchically stored inside the context. For example, the service name is contained in the *ServiceContext* and the operation name in the *OperationContext*. As such, we are able to obtain the aforementioned service id and interface id from the contexts.

Furthermore, these contexts also reserve some empty fields for future extension. We will be making use of these fields for the process id and the sequence id. In particular, in the newly created process id field, we will store the object id of the first service that we encounter as a process id, and pass it on to the next service. Furthermore, Axis2 follows an approach where it will carry over these ids from the previous hop if the ids are available. We will also store the sequence id in one of these reserved slots.

Axis2 integrates Apache Log4j[4] to support logging. The logging configuration can be adjusted by modifying the property file without changing the logging statements in the source code. We can reuse this logging feature to facilitate the information extraction from the Axis2 framework. For instance, we can add logging statements into a handler class, in order to output the information inside contexts. Afterwards we are able to choose which information should be logged and where to store the information by specifying the logging properties. In addition, Axis2 also offers a *SOAPMonitor* module, which is a type of handler, to enable the monitoring of SOAP messages. When a web service built on Axis2 is configured to support this module, the SOAPMonitor is added into the chain of handlers registered by the service inside the Axis2 framework. As result, it will intercept all SOAP messages from/to the web service. This SOAPMonitor can be reused to show the information extracted from the contexts.

[3] `http://axis.apache.org/axis2/java/core/docs/Axis2ArchitectureGuide.html`

[4] `http://logging.apache.org/log4j/`

4. DISCUSSION

In the previous section, we presented the concepts of our approach. As a proof of concept, we have created two simple web services with Axis2 which run on an Apache Tomcat server[5]. The first web service (*WS1*) is concerned with user management, while the second webservice (*WS2*) generates a random number. A user can call WS1 to obtain the user information through a web browser. When WS1 receives the request, it calls WS2 to generate a random number as a user id. Then WS1 sends back the id to the user.

For this preliminary investigation, we made sure to run the web server in a debug mode and used an IDE to dynamically observe each step of the execution. With these observations of the proof of concept setup, we are now in a position to address the research questions that we raised in Section 1.

RQ1: *Which information is required to understand a SOA system?* Basically, in order to understand how the services work together to perform a certain functionality, we need to know how services depend on each other, i.e., how services call each other at runtime. This requires us to have the knowledge of the runtime topology of services. The runtime topology will enable a high-level comprehension of how services interact. A more detailed comprehension process can be started by also tracing the individual services for understanding purposes (e.g., with a tool like Extravis [5]).

RQ2: *How can we exploit the information from a running SOA system?* We addressed this question in Section 2, where an approach for framework-based runtime monitoring is proposed. In particular, we identified the need to collect the following pieces of information: the service's name, namespace and URI, the operation's name and its parameters, the process id and the sequence id. We also instantiated this approach with a prototype implementation in Axis2.

RQ2.1: *If the web service is built on a SOA framework, which information is already inside the framework?* Also in Section 2 we detailed how our approach can be instantiated in the Axis2 framework. Particularly, we can obtain the information for the service id and the interface id straight from the framework. We did add a sequence id and a process id to be able to completely reconstruct the runtime topology. An important follow-up question remains in this context, namely whether our approach generalizes to other SOA frameworks.

5. RELATED WORK

Service monitoring is an essential technique for SOA systems to trace execution, verify regulation, detect runtime

[5] `http://tomcat.apache.org/`

error, etc. Our work focuses more on the understanding of SOA systems through monitoring, and we have identified the following related approaches.

Zmuda et al. [17] propose a flexible monitoring architecture for SOA runtime frameworks. They define requirements for their monitoring architecture and develop a simple proof-of-concept monitoring scenario. Because of the preliminary nature of their work, they discuss technical and implementation issues, but do not introduce concrete monitoring goals.

Li et al. [11] propose an architectural framework to monitor the interactions of web services and validate the interactions with pre-defined constraints at runtime. They intercept the SOAP messages passing through web services, extract important elements corresponding to operation invocation from the content of messages, and then validate the real invocation with interaction constraints. They also focus on the sequence of messages, but mainly in order to try to link a response to the related request.

Baresi et al. monitor the composition of services at runtime [1]. However, they focus on describing the interactions of web services with assertions and checking the conformance between services and their contracts.

Keller et al. [9] present a solution for monitoring service-level agreements for web services. The WSLA framework built in the paper provides an extensible language based on XML schema to define SLAs and a set of monitoring services to automatically monitor the SLAs at runtime.

Another monitoring approach that focuses on fault detection, rather than system understanding, is presented by Chen [3]. They propose a monitoring technique for serviceable component systems with probing algorithms. That is able to flexibly localize the errors happening in the system.

6. FUTURE WORK

In the future, we will continue to carry out our research by extending the current implementation with Axis2 and evaluating our approach. Furthermore, we will consider generalizing the approach to accommodate other service frameworks.

After knowing the runtime topology of a SOA system, which enables a high-level understanding, we aim at extending the current monitoring approach to obtain more information from the SOA system. In particular, we want to extend our approach so that we can:

1. determine at which points in time certain services are scarcely used in order to maintain and test the SOA system at runtime, while keeping the number of users affected by these actions to a minimum.
2. build a statistical user profile, i.e., a profile that contains the most frequently used functionality of a user and the periods at which the user used these. This information can, e.g., be used to improve load balancing.

We also envision to continue our investigation into web service frameworks to determine which other information we can extract from them, and which other functionality we can further reuse for our goals.

In addition, after the information is logged from the system, a next challenge is to efficiently visualize this information, knowing that trace information can sometimes be difficult to scale [4]. We intend to build an application which is able to present the global view of running system. This application would provide a real-time overview of the state of the services and the interactions between them, based on the monitored information.

Acknowledgement. The authors would like to acknowledge NWO for sponsoring this research through the Jacquard ScaleItUp project. Also many thanks to our industrial partners Adyen and Exact.

7. REFERENCES

[1] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *Proceedings of the 2nd international conference on Service oriented computing (ICSOC)*, pages 193–202. ACM, 2004.

[2] G. Canfora and M. Di Penta. Service-oriented architectures testing: A survey. In *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 78–105. Springer, 2009.

[3] Z. Chen. Service fault localization using probing technology. In *Proceedings of the International Conference on Networking, Sensing and Control (ICNSC)*, pages 937–942. IEEE, 2006.

[4] B. Cornelissen, L. Moonen, and A. Zaidman. An assessment methodology for trace reduction techniques. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, pages 107–116. IEEE, 2008.

[5] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.

[6] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[7] C. J. Fidge. Timestamp in message passing systems that preserves partial ordering. In *Proceedings of the Australian Computing Conference*, pages 56–66, 1988.

[8] N. Gold, C. Knight, A. Mohan, and M. Munro. Understanding service-oriented software. *IEEE Software*, 21(2):71–77, 2004.

[9] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

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

[11] Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 70–79. IEEE CS, 2006.

[12] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.

[13] J. Moe and D. A. Carr. Understanding distributed systems via execution trace data. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 60–67. IEEE CS, 2001.

[14] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer. Service-oriented computing: A research roadmap. In F. Cubera, B. J. Krämer, and M. P. Papazoglou, editors, *Service*

Oriented Computing (SOC), number 05462 in Dagstuhl Seminar Proceedings, 2006.

[15] J. Snell, D. Tidwell, and P. Kulchenko. *Programming Web Services with SOAP*. O'Reilly Media, 2001.

[16] A. Zaidman, M. Pinzger, and A. van Deursen. Software evolution. In P. A. Laplante, editor, *Encyclopedia of Software Engineering*, pages 1127–1137. Taylor & Francis, 2010.

[17] D. Zmuda, M. Psiuk, and K. Zielinski. Dynamic monitoring framework for the SOA execution environment. *Procedia Computer Science*, 1(1):125–133, May 2010.