Understanding Service-Oriented Systems Using Dynamic Analysis

(Position paper)

Tiago Espinha
Delft University of Technology
The Netherlands
Email: t.a.espinha@tudelft.nl

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

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

***Abstract*—When trying to understand a system that is based on the principles of Service-Oriented Architecture (SOA), it is typically not enough to understand the individual services in the architecture, but also the interactions between the services. In this paper, we present a technique based on dynamic analysis that can be used to obtain insight into how services work together to perform overall business functionality. In particular, our technique connects traces from individual services together, so that the user can obtain a global understanding of how the entire SOA works.**

I. Introduction

The fact that software systems need to evolve in order to remain successful has been long recognized [1]. With the advent of Service Oriented Architectures, the maintenance problem was said to become easier [2], as a Service Oriented Architecture (SOA) would be composed of several loosely coupled (and smaller) services. However, in 2004 already, Gold et al. warned that even though the actual maintenance was possibly easier, the understanding of SOA-based systems would still be a major and costly challenge [2]. This is mainly due to the shift from understanding a monolithic application to understanding a distributed system composed of many entities (services).

Thus, program comprehension remains an absolute and important prerequisite to maintaining systems [3], including systems built around the SOA concept. In this context Corby even reported that up to 60% of the time effort of a maintenance task lies in understanding the system at hand [4]. With documentation missing or being obsolete, *reverse engineering* is often a viable option [5] for reconstructing (part of) the documentation, which, in turn, enables the understanding of the system.

In the context of this research, we choose a reverse engineering approach based on dynamic analysis. Our choice is instigated by the fact that we expect dynamic analysis to better cope with the dynamic nature of service-oriented systems, in which services can easily be exchanged for other services. With the dynamic analysis approach, we can collect runtime data in order to obtain knowledge about the system under scrutiny [6].

While extensive research has been done in the area of dynamic analysis for program understanding, most of the research has focused on analyzing and understanding monolithic systems [6]. In contrast, our work aims to alleviate the understanding of complex, distributed systems.

We specifically focus on systems built around the SOA paradigm and technologies such as SOAP[1] and REST[2]. While tracing and understanding each of the individual services in a SOA is comparable to working with monolithic versions, there is a challenge in understanding how different services (dynamically) work together to deliver functionality. This challenge is further complicated by the fact that communication between services might happen in an asynchronous way, calling for solutions that bring clarity to the situation (similar to the case of asynchronous communication in Ajax web applications, e.g. [7]).

Our research for alleviating the pains of understanding complex SOA based software systems is steered by the following research questions:

- **RQ1** How can we obtain a trace of requests as they traverse several web services and which data should be included?
- **RQ2** How can dynamic analysis of a distributed system further aid its understanding?

II. Terminology & Scenario Description

In order to further explain the foundation of our research, we first clarify in Section II-A some of the terminology that we are going to use. Subsequently, we describe an actual scenario description in which our approach can be useful (see Section II-B).

A. Clarification of Terminology

It is common practice to refer to any SOAP-enabled component as a web service but this terminology induces confusion. In fact, the W3C Working Group defines the term web service as "a software system designed to support interoperable machine-to-machine interaction over a network"[3]. It is also a fact that two major technology platforms (Java and C#) currently use a *web service* as a container for several

[1]Simple Object Access Protocol – http://www.w3.org/TR/soap/

[2]REpresentational State Transfer – https://www.ibm.com/developerworks/webservices/library/ws-restful/

[3]Web Services Glossary - http://www.w3.org/TR/ws-gloss/

web methods. These web methods (sometimes referred to as operations) can then be remotely executed via a SOAP request.

For this paper, we will follow the notion of a web service as a container for web methods which are in turn the simplest code-bearing units.

B. Scenario Description

For the scope of this paper, we have developed a small proof of concept system composed of six SOAP web services that have pre-defined (although loosely coupled) associations (Fig. 1). The loose coupling implies that the web services involved might or might not be deployed in the same infrastructure; it also means that several instances of the same web service may be deployed across different servers for load balancing purposes.

In this system, the composing web services perform the following tasks:

- **MD5WS** and **SHA1WS**: each contain a web method `getHash` which takes a string and returns the respective MD5 or SHA1 hash.
- **HashWS**: contains a web method `getHash` which takes two strings: the string to be hashed and the type of hashing to perform (MD5 or SHA1).
- **RandWS**: contains a web method `getRandom` which returns randomly generated integers (from Java's Integer.MIN_VALUE to Integer.MAX_VALUE).
- **SumWS**: contains a web method `add` which calculates and returns the sum of two arguments.
- **OrchestratorWS**: contains two web methods. The first, `orchestrate` takes two integers and a string specifying which hashing method to use, therefore forsaking the usage of the `getRandom` web method. The second web method, however, is fully automated and requires no arguments.

The interactions between these web services are depicted by arrows in Figure 1. While this sample system is small and its interactions are known in advance, our goal is to demonstrate that the interactions can be inferred from the trace data generated by our solution. In this case, considering we have detailed knowledge of the system being analyzed, we can easily verify our findings by comparing them to how the system is in fact laid out.

III. Tracing Approach

The challenge with performing dynamic analysis on a monolithic system lies in visualizing the trace data that is readily available through the use of debugging or logging tools. When it comes to distributed systems, it is still equally easy to produce execution data for each server. However, as far as we know, there currently exists no way of *connecting* the individual traces from services in order to get a global view of how they work together to deliver a specific piece

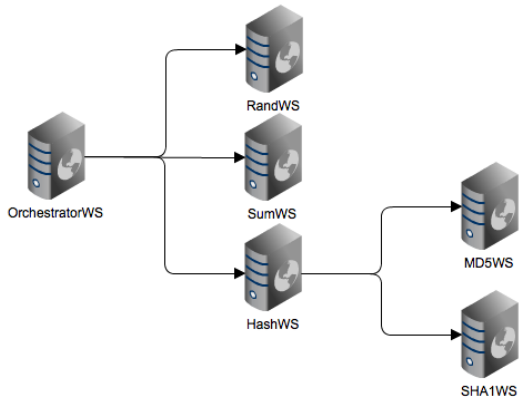

Figure 1. Scenario

of functionality (also see Cornelissen's systematic literature survey [6]).

For instance, when dealing with large quantities of simultaneous requests (as a distributed system is meant to), it is impossible to determine that an execution trace *t1* of a call to *MD5WS.getHash* was in fact triggered by a call to *HashWS.getHash* (with its respective trace *t2*) and not by an entity external to our system.

In order to achieve this, we propose adding a tag to each SOAP request which will allow us to identify causality in the trace logs (e.g. trace t2 happened as a result of trace t1) of our distributed system.

Since the SOAP protocol relies on lower level HTTP to transport the SOAP message, we can use one of the existing HTTP request fields that plays no major role in SOAP in order to add the tag in a seamless manner. By doing so, we ensure that the HTTP stack requires no changes thus creating no deviation from the original standard. A good candidate for this role is the *user-agent* field[4]. The sole purpose of this field is to provide the web server with information about the client (e.g. version, HTML rendering engine, etc) and it should not be used as a decision-influencing field for web services' business logic. For this reason, we can safely append (or eventually replace altogether) the default user-agent string by a tag that allows us to follow a trail of web method calls.

We can establish this trace by (creating and) placing a request identifier on the user-agent field whenever one is not already present. If an identifier already exists, then it is copied into all the outgoing web method invocations. Whenever an external SOAP client invokes a web method under the domain of our instrumented system, it will carry no request identifier. However, if a web method *m1* within our system invokes another web method *m2* either within the same or a different web service, then the outgoing SOAP request will be modified to include the request identifier created upon the invocation of *m1*.

[4]RFC 2616, Chapter 14.43 http://www.ietf.org/rfc/rfc2616.txt

IV. Implementation Challenges and Compromises

In this section we present the technical challenges in performing dynamic analysis on a distributed SOAP-based system. In subsections IV-A and IV-B we discuss our technical approach through the use of AOP and in subsections IV-C and IV-D we reason about the granularity of data that is stored as a part of the tracing.

A. Using Aspects For Tracing In Java

The greatest challenge in establishing a link between the traces of web methods that invoke each other, lies in carrying the request identifier from the entry point of the web method (its invocation) into all the possible exit points (this web method invoking other remote web methods).

This is particularly difficult to achieve, as the action of parsing an incoming SOAP request is independent of creating further outgoing SOAP requests. In fact, a web method does not necessarily invoke other web methods; as is the case with RandWS, SumWS, MD5WS and SHA1WS in our proof of concept. Since such a feature does not exist in the protocol itself and the most prominent frameworks also do not possess this capability, we have devised a way of making this possible by using aspect-orientation techniques (AOP).

Aspect-orientation allows us to weave additional functionality into the code through *aspects* [8]. For the purpose of our research, we have focused on aspect-orientation in Java as this technique is well supported in Java through AspectJ[5].

To achieve our goal of tracing, we have defined three distinct aspects:

1. `JAXAspect.aj` is woven into the JAX-WS[6] runtime library (jaxws-rt.jar) and intercepts all incoming SOAP requests in a non-obtrusive way. Whenever a request is received, the advice for this aspect checks whether there is a request identifier in the user-agent field and does one of two things. If a request identifier exists, it means the request is coming from a web method instrumented with our aspects; because of this, the aspect stores the existing request identifier in a ThreadLocal integer. If a request identifier is not present in the user-agent field, then the aspect will poll a static AtomicInteger (chosen for thread-safety reasons) for the current integer, whilst incrementing it by 1.

2. `JDKAspect.aj` is woven into Java's JDK itself (we used Sun's latest JDK, 1.6.0_23-b05) and is responsible for intercepting (also non-obtrusively) all the outgoing SOAP requests. Whenever the advice for this aspect is being executed, we know that an outgoing SOAP request is being sent by one of the web methods in our system. More importantly, we know that there has been an incoming request beforehand and thus, we have a request identifier stored in the ThreadLocal variable. For this reason, we simply need to replace the user-agent field with the stored request identifier and then allow the request to flow as it would have had otherwise.

3. `CatchAllAspect.aj` is a simple optional aspect that is automatically deployed on all web service containers (.war archives) and it offers a more fine-grained trace of the system. Essentially, it logs all method calls for every class contained in the web service archive. In Section IV-D we provide a discussion of the benefit obtained from this aspect versus what it means regarding performance.

B. Logging Trace Data

While there are many reliable tools that can be used for logging, it was our goal to propose a lightweight toolset that could be deployed in any Java/JAX-WS system with little effort and without affecting possible existing logging facilities. For this reason, we created a fourth aspect (**TomcatAspect.aj**) which, as the name suggests, is meant to be deployed on the underlying Tomcat server. This aspect spawns an arbitrary number of separate threads whose only purpose is to deal with logging requests. By funneling all the logging into purposely created threads (via a LinkedBlockingQueue), we ensure that our logging is non-blocking for the execution of the actual web methods, regardless of the underlying storage technology that is being used.

In our study, the logger threads dispatch the log requests to a remote server via an HTTP POST request to a PHP script which stores the data in a relational database system.

We have also found that the number of spawned threads influences the celerity with which log requests are processed. Fewer threads resulted in a backlog of logging requests even after the actual SOAP requests had ceased. Eventually, the amount of threads could change dynamically to account for load peaks on the server, but this falls outside the scope of this paper and will be addressed in further research.

Also relevant questions for logging are *when and what to log?* These two factors will condition how much information we can leverage out of the system and how it will impact the overall performance in terms of CPU usage and network bandwidth.

C. What to log?

Effectively, the question of *what* to log is an important one. Logging higher amounts of trace data means being able to obtain finer grained information about the system, but requires more network bandwidth and storage space. On the other hand, logging too little data might also defeat the purpose of tracing altogether, by not providing enough insight into the runtime behavior of the system.

In our research we attempted to log the data that seemed necessary to obtain a global view of the system in study. We did this without regard for the amount of bandwidth

[5]AspectJ - http://www.eclipse.org/aspectj/

[6]JAX-WS - http://jax-ws.java.net/

or storage space required. Establishing a trade-off between the amount of data obtained and real-world limitations is an open question we would like to leave for further research.

Following is a list of concrete data our toolkit collects from the system, and the rationale behind the relevance of each datum.

Timestamp: Each log entry contains a temporal reference, as defined by the party making the log request. As mentioned in Section IV-D, this provides essential information regarding execution times and network delays. More concretely, we are able to identify network bottlenecks (i.e. certain network links with high delay or reaching its maximum throughput) and web methods that take too long to execute.

Web service/method signature: These two identifiers allow us to unequivocally match each log entry to a specific piece of code (i.e. web method). Since different web services can have web methods with the same name, it is also equally important to store the name of the web service.

IP address of log-requester: Since we log different types of events (incoming and outgoing), it could be confusing to store different types of IP addresses (for example, do we store the request sender's IP or the receiver's?). For this reason, we decided to store the IP address of the system that is generating the log request. This information can be gathered on the log-server's side thus also reducing strain on the network.

D. When to log?

The question of *when* to log also influences the granularity of the information that we want to leverage from the system with our analysis. For our research, we established two levels of event granularity: (1) a coarse-grained level where only the web method calls are logged, in a black-box manner and (2) a fine-grained level where the web methods are fully instrumented and all method calls are logged.

Web methods only. In the more coarse-grained level we log only the incoming and outgoing calls to web methods, disregarding the actual code that is ran inside. This means logging whenever a web method is invoked, but also when the web method has finished its execution.

These two temporal references give us an important statistical value regarding the execution times for each web method. If a web method consistently takes too long to return, we can then attempt to identify the culprit for the response delay; since we log running time data for all the web methods within our system, we can then select log details with the same request identifier and determine which web method is causing the bottleneck.

Still at the coarse-grained level, we refined the granularity further by logging both the event of a client sending a request as well as that of the web service receiving it. This added dimension provides insight into an eventual network congestion. Execution delays caused by the network can be detected whenever the timestamp of an incoming request from a server's point of view is much greater than that of when the client made the request. This is a factor that only becomes relevant whenever distributed systems are concerned, as all the communication happens over a network which is delay-prone.

Full instrumentation. The other, more fine grained, approach is incremental and builds upon the previous approach. It consists of logging all method calls done inside each web method. Obviously, we expect the more fine-grained approach to have a higher performance impact, but the trade-off is obtaining more runtime information about the system.

For instance, with this added granularity we can determine not only which web method is causing a bottleneck but also which method calls within that web method are responsible for this delay.

V. Log Analysis

No less important than collecting data, is being able to transform it into useful information. The approach presented in the previous sections collects runtime trace data into a central repository but it is also important to understand whether the objectives of our research are met.

Following is a trace as it is stored in the log server:

ID	Tstamp (ns)	Web method signature	in/out
1	3289898871267858	{http://sam.t/}orchestrateAuto	i
1	3289899175622484	t.sam.OrchWSImpl	o
1	3289899203395842	{http://maths.t/}getRandom	i
1	3289899223786466	t..sam.OrchWSImpl	o
1	3289899225207985	{http://maths.t/}getRandom	i
1	3289899255149777	t.sam.OrchWSImpl	o
1	3289899256819840	{http://webservices.t/}add	i
1	3289899290090600	t.sam.OrchWSImpl	o
1	3289899292200837	{http://sec.tudelft/}getHash	i
1	3289899341804535	t.sec.HashWSImpl	o
1	3289899343719864	{http://md5.sec.t/}getHash	i

Table I
Trace Sample

In Table I we have all the requests with ID #1, sorted by their timestamp. By analyzing the table, we can determine that OrchWSImpl.orchestrateAuto() is the first SOAP request to be placed on this trace. In turn, it places two outgoing calls to the getRandom() web method contained in another web service. After two random numbers are obtained, their sum is calculated through a call to the *add* web method and the result of the sum is then hashed via the *getHash* web method in the HashWSImpl web service. This last web service calculates either an MD5 or SHA1 hash, depending on the arguments it receives. In the case at hand, we can see that the *getHash* web method being executed is contained in the package *t.sec.md5*, thus, the hashing algorithm used was MD5.

This table also contains information about network delays and web method running times. For instance, calculating the difference between the timestamps in the second and

third lines of the table (an outgoing and an incoming event, respectively), we obtain the network delay for this request (∼27.8ms for this particular case). On the other hand, to obtain information about web method running times, the full instrumented approach should be used as it provides more granularity on the method calls.

VI. Discussion & Future Challenges

In the introduction of this paper, we presented two main research questions that we attempted to address. These questions help us in studying the underlying research question of how we can enable the understanding of service-oriented systems. To this effect and referring to *RQ1*, our approach described in Section III does indeed make it possible to trace related SOAP requests as they traverse several web methods.

For *RQ2*, which is addressed in Section V, we demonstrate how the dynamic analysis approach can be used to outline the interactions of the system. This approach still has more to offer in its fully instrumented format but this is a topic we would like to explore in future work.

A. Future work

This position paper describes the foundations of an approach for enabling the understanding of service-oriented systems. With these foundations now in place we aim to take the following steps in follow-up research:

1) **Performance.** As our approach adds functionality (and thus, lines of code) to the system, a performance impact is to be expected [9]. As performance is an important concern for our research as we want to validate the applicability of this approach for an eventual industrial scenario, we aim to investigate the performance penalty of our approach. Preliminary tests seem to show that the performance impact is small, but we need to investigate this further in order to be able to draw firm conclusions.
2) **Creating a "dashboard".** In order to stimulate the understanding of complex systems, we aim to provide a visual representation of the system under study. This so-called *dashboard* would enable to track requests coming into the system and following them as they travel through the system.
3) **User study.** In order to evaluate the usefulness of our approach and the dashboard, we aim to perform a user study (e.g, [10]), in which a number of developers use our toolset to understand a service-oriented system.

VII. Related Work

Based on the systematic literature survey in the area of program comprehension through dynamic analysis from Cornelissen et al. we have created a short overview of related work [6].

Briand et al. worked on generating sequence diagrams from distributed software systems. Their approach was centered around the Java Remote Method Invocation (RMI) technology [11].

Edwards et al. presented an approach to perform feature location in distributed systems [12]. Key to their approach is that they use timings of components to determine the order of execution.

Moe and Sandahl use dynamic analysis to understand software systems built around the CORBA middleware platform [13].

Acknowledgment

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.

References

[1] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*, ser. Apic Studies In Data Processing. Academic Press, 1985.

[2] N. Gold, C. Knight, A. Mohan, and M. Munro, "Understanding service-oriented software," *IEEE Software*, vol. 21, no. 2, pp. 71–77, 2004.

[3] A. Zaidman, M. Pinzger, and A. van Deursen, "Software evolution," in *Encyclopedia of Software Engineering*, P. A. Laplante, Ed. Taylor & Francis, 2010, pp. 1127–1137.

[4] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[5] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[6] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Software Eng*, vol. 35, no. 5, pp. 684–702, 2009.

[7] N. Matthijssen, A. Zaidman, M.-A. Storey, I. Bull, and A. van Deursen, "Connecting traces: Understanding client-server interactions in ajax applications," in *Proc. 18th Int. Conf. on Program Comprehension (ICPC)*. IEEE CS, 2010, pp. 216–225.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 1241. Springer, 1997, pp. 220–242.

[9] A. Zaidman and S. Demeyer, "Automatic identification of key classes in a software system using webmining techniques," *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, vol. 20, no. 6, pp. 387–417, 2008.

[10] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 341–355, 2011.

[11] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of UML sequence diagrams for distributed Java software," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 642–663, 2006.

[12] D. Edwards, S. Simmons, and N. Wilde, "An approach to feature location in distributed systems," *J. Syst. Software*, vol. 79, no. 1, pp. 457–474, 2006.

[13] J. Moe and K. Sandahl, "Using execution trace data to improve distributed systems," in *Proc. Int. Conf. on Software Maintenance (ICSM)*. IEEE CS, 2002, pp. 640–648.