Delft University of Technology
Software Engineering Research Group
Technical Report Series

# Extracting Dynamic Dependencies between Web Services Using Vector Clocks

Daniele Romano, Martin Pinzger and Eric Bouwers

Report TUD-SERG-2011-034


TUD-SERG-2011-034

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of the International Conference on Service-Oriented Computing and Applications (SOCA), 2011, IEEE Computer Society.

# Extracting Dynamic Dependencies between Web Services Using Vector Clocks

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

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

Eric Bouwers
*Software Improvement Group*
*Amsterdam, The Netherlands*
*Email: e.bouwers@sig.eu*

***Abstract*—Service Oriented Architecture (*SOA*) enables 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 belonging to different departments of an organization.**

**In this paper, we propose an approach to extract dynamic dependencies among web 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.**

**We show the feasibility of the approach by implementing it into the Apache CXF framework and instrumenting the SOAP messages. We designed and executed two experiments to investigate the impact of the approach on the response time. The results show a slight increase that is deemed to be low in typical industrial service oriented systems.**

***Keywords*-SOA; web services; dynamic dependencies;**

## I. Introduction

IT organizations need to be agile to react to changes in the market. As a consequence they started to develop their software systems as Software as a Service (*SaaS*), overcoming the poor inclination of monolithically architected systems towards *agility*. Hence, the adoption of Service Oriented Architectures (*SOAs*) has become popular. In addition, SOA-based application development also aims at reducing development costs through service reuse.

On the other hand, mining dependencies between services in a *SOA* is relevant to understand the entire system and its evolution over time. The distributed and dynamic nature of those architectures makes this task particularly challenging.

In order to get an accurate picture of the dependencies within a SOA system a dynamic analysis is required. Using static analyses simply fails to cover important features of a SOA architecture, for example the ability to perform dynamic binding. To the best of our knowledge, existing technologies used to deploy a service oriented system do not provide tool to accurately detect 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, such as HP OpenView SOA Manager[3], allow the exploration of the dependencies, but they must explicitly be specified by the user [1].

In this paper, we propose (1) an adaptation of our approach based on *vector clocks* [14] to extract dynamic dependencies among web services deployed in an enterprise; (2) a non-intrusive, easy-to-implement and portable implementation and (3) an analysis of the impact of our approach on the performance.

*Vector clocks* have originally been conceived and used to order events in a distributed environment [8], [4]. We bring this technique to the domain of service oriented systems by attaching the *vector clocks* to SOAP messages and use them to order service executions and to infer causal dependencies.

The approach has been implemented into the Apache CXF[4] framework taking advantage of the *Pipes and Filters* pattern [5]. Since this pattern is widely used in the most popular web service frameworks and Enterprise Service Buses, the approach can be implemented on other SOA platforms (*e.g.*, Apache Axis2[5] and Mule ESB) in a similar manner.

To analyze the impact of the approach on the performance of a system we investigate how the approach affects the response time of services. The results show a slight increase due to the increasing message size and the instrumented Apache CXF framework. To determine the impact on real systems a repository of 41 industrial systems is examined. Given the amount of services typically deployed within this industrial systems we do not expect a significant increase of the response time when using our approach.

This paper is structured as follows. In Section II we present the main applications of the proposed approach. In

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

Section III we report the related work. In Section IV we describe the context in which we plan to apply our study. In Section V we describe our approach to extract dynamic dependencies among web services. In Section VI we propose an implementation of our approach. In Section VII we report the first experiments and the obtained results. Finally, we conclude the paper and present the future work in Section VIII.

## II. Applications

In this section we discuss the main applications of our approach that we plan to perform in future work.

### A. Quality attributes measurement

Our approach can be used to build up dynamic dependency graphs. These graphs are commonly weighted, where the weights indicate the number of times a particular service is invoked or a particular execution path is traversed. The information contained in these graphs can help software engineers to measure important quality attributes (*e.g.*, analyzability and changeability) for measuring maintainability of the system under analysis.

For instance Perepletchikov *et al.* defined several cohesion and coupling metrics to estimate the maintainability and analyzability of service oriented systems [10], [11], [12], [9]. In our previous work [13], we found an interesting correlation between the number of changes in Java interfaces and the external cohesion metric defined by Perepletchikov *et al.* With our approach to extract dynamic dependencies among services we plan to perform similar studies to validate and improve those metrics by analyzing service oriented systems. More in general, our dynamic dependency analysis is a starting point to study the interactions among services in industrial service oriented systems and to define anti-patterns that can affect the quality attributes required by a *SOA*.

### B. Change Impact Analysis

Besides the measurement of quality attributes our approach can be used to perform Change Impact Analyses (IA) on service oriented systems. Bohner *et al.* [15], [2] defined the IA as the identification of potential consequences of a change, or the assessment of what needs to be modified to accomplish a change. They defined two techniques to perform IA, namely *Traceability* and *Dependency*.

Wang *et al.* [16] defined an IA approach for service oriented systems based on a service dependency graph. Our approach fits in with their work by adding a dynamic dependency graph.

## III. Related Work

The most recent work on mining dynamic dependencies in service oriented systems 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. This approach does not require the instrumentation of the system infrastructure. However, it is based on probabilities and there is still the need for properly setting the parameters of their algorithms to reach a good accuracy.

In 2006, Briand *et al.* [3] 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 which in turn requires a complete knowledge of the system.

Hrischuk *et al.* [6] provided a series of requirements to reverse engineer scenarios from traces in a distributed system. However, besides the requirements, this work does not provide any approach to extract dependencies in a service oriented system.

As can be deduced from the overview of related work there currently does not exist any accurate approach for inferring the dependencies amongst services. In this paper, we present such an approach based on the concept of *vector clocks*.

## IV. Study Context

In this section we describe the context in which we plan to apply our study. The perspective is that of a quality engineer who wants to extract the dynamic dependencies among services within the boundaries of an enterprise. We refer to dependencies as message dependencies, according to which two services are dependent if they exchange messages. We furthemore refer to web services as services which are compliant to the following XML-standards:

- WSDL[6] (Web Services Description Language) which describes the service interfaces.
- SOAP[7] (Simple Object Access Protocol) widely adopted as a simple, robust and extensible XML-based protocol for the exchange of messages among web services.

Finally, we assume that the enterprise provides an UDDI[8] (Universal Description, Discovery, and Integration) registry to allow for the publication of services and the search for services that meet particular requirements.

Our sample enterprise is composed of several departments (a sample enterprise with two departments is shown in Figure 1). Each department exposes some functionality as web services that can be invoked by web services deployed in other departments. Services deployed within the boundaries of the enterprise are called *internal services*. Services deployed outside the boundaries of the enterprise are called *external services*.

[6] http://www.w3.org/TR/wsdl
[7] http://www.w3.org/TR/soap/
[8] http://uddi.xml.org/

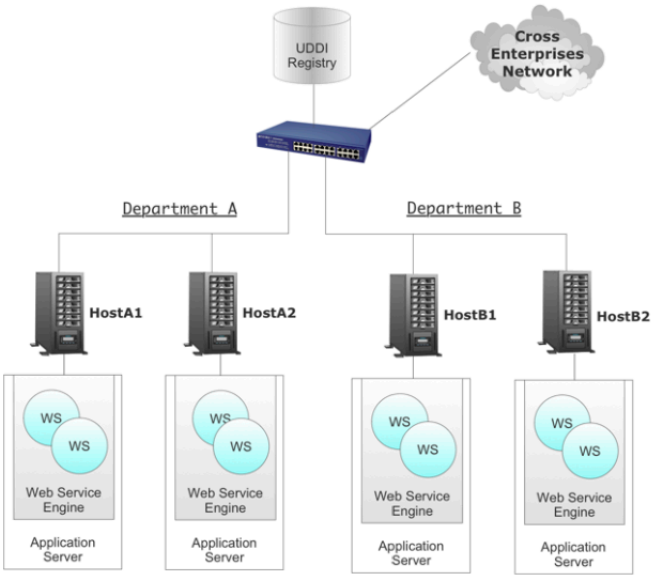


Figure 1. A sample enterprise with web services deployed in two departments

We assume that hosts within the departments publish web services through an application server (e.g., *JBoss AS*[9] or *Apache Tomcat*[10]) and web service engines (*e.g.*, Apache Axis2 or Apache CXF).

## V. Approach

Our approach to extract dynamic dependencies among web services is based on the concept of *vector clocks*. In this section, we first provide a background on *vector clocks* after which we present our approach to order service executions and to infer dynamic dependencies among web services.

### A. Vector Clocks

Ordering events in a distributed system, such as a service oriented system, is challenging since the physical clock of different hosts may not be perfectly synchronized. The logical clocks were introduced to deal with this problem. 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.

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.

### B. Inferring dependencies among web services

We conceive a *vector clock* (*VC*) as a vector/array of pairs *(s,n)*, where *s* is the service id and *n* is number of times the service *s* is invoked. When an instance of the service *s* receives an execution request the *vector clock* is updated according to the following rules:

- if the request does not contain a *vector clock* (*e.g.*, a request from outside the system), the *vector clock* is created, and the pair *(s,1)* is added to it;
- if the request contains a *vector clock* and a pair with service 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.

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 then stored in a database.

Whenever an instance of the service *s* sends an execution request to another service *x*, then the following actions are performed:

- if the service *x* is an *internal service*, then the *vector clock* is attached to the outgoing message;
- if the service *x* is an *external service*, the pair *(x,1)* is added to the *vector clock* and the *vector clock* is stored in the database but not attached to the outgoing message.

From the set of *vector clocks* stored in the database, we can infer the causal order of the service executions. Given the *vector clocks* associated to the execution of the service *i* and the service *j*, *VC(i)* and *VC(j)*, we can state that the execution of service *i* causes the execution of 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 for *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 values for *n* that are less or equal to the corresponding values for *n* in *VC(j)*, and at least one value for *n* is smaller.

[9] http://www.jboss.org/jbossas/
[10] http://tomcat.apache.org/

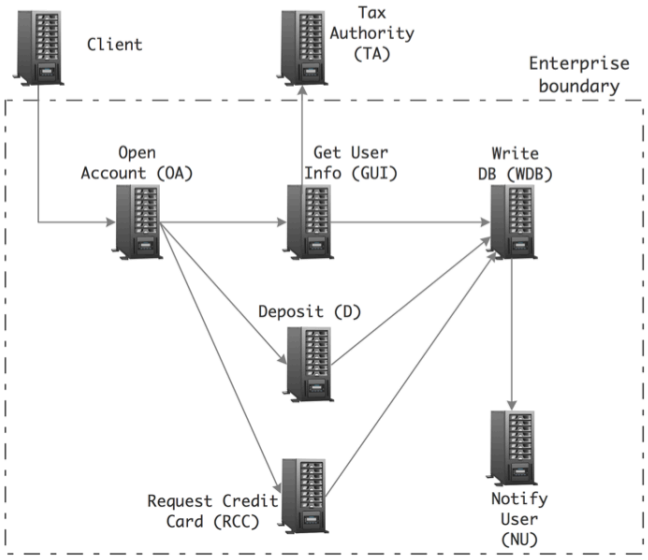


Figure 2. Example of a service oriented system to open a bank account

If all the corresponding pairs of the two *vector clocks VC(i)* and *VC(j)* contain the same values for *n* except one corresponding pair whose values for *n* differ exactly by 1, we state that there is a direct dependency (*i.e.*, a direct call) between service *i* and service *j*.

If a pair with id *s* is missing in the vector the value for *n* is considered to be 0.

Finally, to infer the dynamic dependencies among services, it is necessary to apply the binary relation in (1) among each pair of *vector clocks* whose values are stored in the database.

### C. Working Example

Consider the example system from Figure 2 composed by six services inside the enterprise boundary, one external service and one client which triggers the execution. The system provides the services to open an account in a banking system.

In this example, the client interested in creating an account needs to invoke the service *OpenAccount*. This service invokes the services *GetUserInfo*, *Deposit* and *RequestCreditCard*. These services invoke the service *WriteDB* to access a database. *WriteDB* first writes in a database and then, if its invocation has been triggered by *RequestCreditCard*, invokes *NotifyUser* which performs actions to notify the user. The external service *TaxAuthority* is invoked by *GetUserInfo* to inquire fiscal information about the user.

The execution flow resulting from the invocation of the service *OpenAccount* is shown as a UML sequence diagram in Figure 3. The arrows in the diagram are labeled with the *vector clocks* associated to the execution of the invoked service. *Vector clocks* with superscripts mark *vector clocks* associated to different instances of the same service. When the *OpenAccount (OA)* service is invoked, there is no *vector clock* attached to the message, since the invocation request comes from outside (*i.e.*, Client). Hence, a new *vector clock* (*VC(OA)*) is created with the single pair *(OA,1)* and it is stored in the database. Then the execution of the service *OpenAccount* triggers the execution of the service *GetUserInfo (GUI)*. When this service is invoked, a new pair *(GUI,1)* is added to the *vector clock*, obtaining the new clock *VC(GUI)=[(OA,1),(GUI,1)]* that is stored in the database.

When the service *GetUserInfo (GUI)* invokes the external service *TaxAuthority (TA)* the *vector clock* is set to *VC(TA)=[(OA,1),(GUI,1),(TA,1)]* and is stored in the database. In this way we can infer dependencies to external services. Since *TaxAuthority (TA)* is an external service and we do not have control of external services the *vector clock* is not attached to this message.

Consider the execution of the service *WriteDB (WDB)*, and assume we want to infer all the services that 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 clocks VC(x)* satisfy the following boolean expression:

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

These services are *OpenAccount*, *GetUserInfo*, *Deposit* and *RequestCreditCard* (see Figure 3).

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:

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

The sole service which *WriteDB* depends on is *NotifyUser*.

Consider the execution of the service *OpenAccount (OA)*, and assume we want to infer the services that *OpenAccount* depends on directly. Those services are the services *GetUserInfo(GUI)*, *Deposit(D)* and *RequestCreditCard(RCC)*. Their *vector clocks* (*VC(GUI)*, *VC(D)* and *VC(RCC)*) contain only one pair (respectively *(GUI,1)*, *(D,1)* and *(RCC,1)*) with a value for *n* that is larger exactly by 1 than the corresponding values in the *vector clock VC(OA)*. Among the services *OA* and *WDB* there are no direct dependencies because the *vector clocks* corresponding to the execution of *WDB* contain two pairs with different values for *n*.

The values for *n* from the example in Figure 3 are all equal to 1. However, they are needed to detect the presence of cycles along the execution flows. Assume that the *NotifyUser* service invokes the *WriteDB* service introducing a cycle. In this case the *vector clock* associated to the second invocation of the service *WriteDB* is *VC(WDB)=[(OA,1),(RCC,1),(WDB,2),(NU,1)]*.

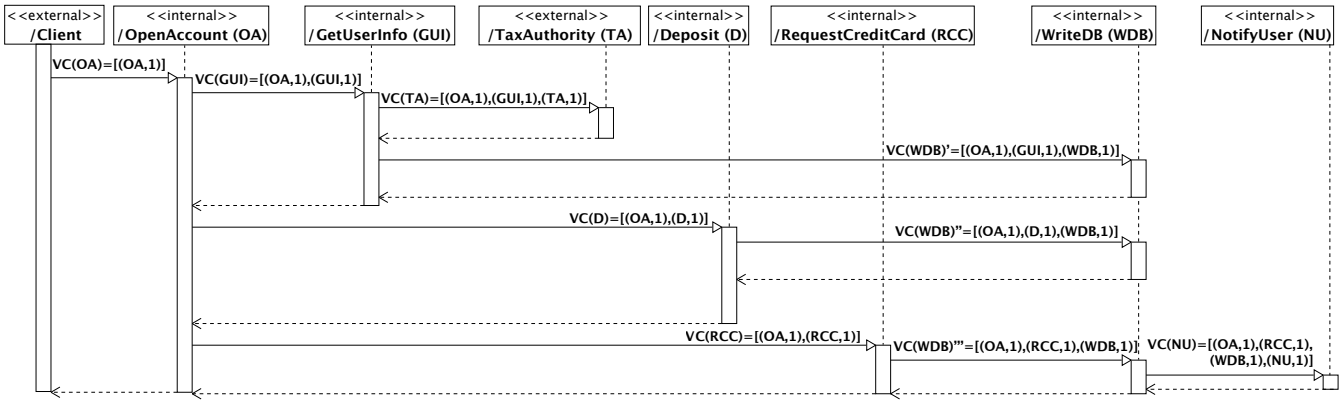


Figure 3. Sequence diagram for opening a bank account. The arrows in the diagram are labeled with the *vector clocks* associated to the execution of the invoked service.

## VI. Implementation

The implementation of the proposed approach should be non-intrusive, easy-to-implement and portable to different SOA platforms. Only if these properties hold we can be sure that the approach can be adapted in an industry setting. In this section we propose an implementation that meets these requirements.

The implementation requires three steps. First, the messages need to be instrumented to attach the *vector clock* data structure. Next, we need a technique to capture the incoming messages in order to retrieve the *vector clock*, update it and store its value in the database. Finally the outgoing messages have to be captured to attach the updated *vector clock* to them.

To instrument the messages we use the SOAP header element. This element is meant to contain additional information (*e.g.*, authentication information) not directly related to the particular message.

For example, after attaching the *vector clock* to the message sent from the service *GetUserInfo* to the service *WriteDB* (see Figure 3), the message contains the following header:

```
<soap:Envelope>
<soap:Header>
  <vc:VectorClock>
    <vc:pair>
        <vc:s>OpenAccount</vc:s>
        <vc:n>1</vc:n>
    </vc:pair>
    <vc:pair>
        <vc:s>GetUserInfo</vc:s>
        <vc:n>1</vc:n>
    </vc:pair>
  </vc:VectorClock>
</soap:Header>
...
</soap:Envelope>
```

Concerning the interception of the incoming and outgoing messages, we adopted a technique that relies on the *Pipes and Filters* [5] architectural pattern. The *Pipes and Filters* pattern allows to divide a larger processing task into a sequence of smaller, independent processing steps, called *Filters*, that are connected by channels, called *Pipes*. This pattern is widely adopted to process incoming and outgoing messages in web service engines and frameworks such as Apache Axis2 and Apache CXF.

Those frameworks use *Filters* to implement the message processing tasks (*e.g.*, messages marshaling and unmarshaling) and they allow the developers to easily extend the chains of *Filters* to further process messages. Since this pattern is widely used, even by the Enterprise Service Bus platforms (*e.g.*, MuleESB), we decided to use this pattern to implement the logic needed to retrieve, update, store and forward the *vector clocks*.

Instrumenting the services would be an alternative implementation approach. However, instrumentation is risky since it modifies the implementation and can introduce bugs. To implement our approach we use the Apache CXF service framework. In Apache CXF the *filters* are called *interceptors*. Figure 4 shows the chains of interceptors between an *Apache CXF Deployed Service* and an *Apache CXF Developed Consumer*.

When the consumer invokes a remote service, the Apache CXF runtime creates an outbound chain (*Out Chain*) to process the request. If the invocation starts a two-way message exchange, the runtime creates an inbound chain to process the response (omitted in Figure 4).

When a service receives a request from a consumer, a similar process takes place. The Apache CXF runtime creates an inbound interceptor chain (*In Chain*) to process the request. If the request is part of a two-way message exchange, the runtime also creates an outbound interceptor chain (omitted in Figure 4).

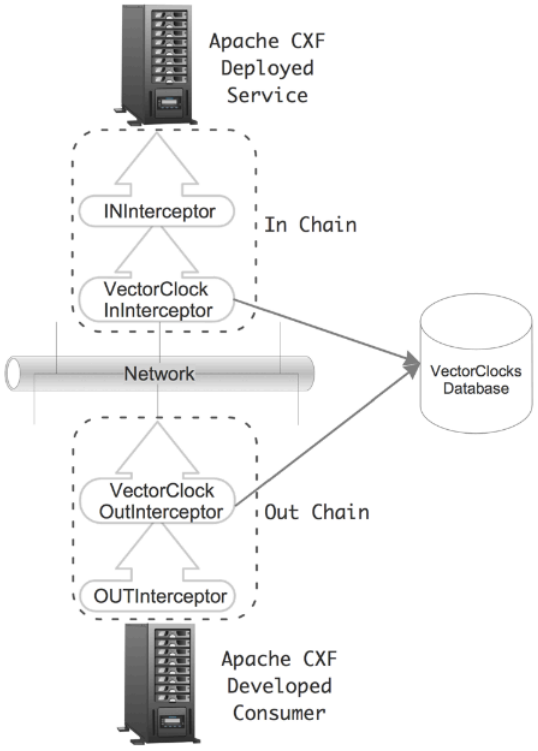


Figure 4. The chains containing our vector clock interceptors between a *Apache CXF Deployed Service* and a *Apache CXF Developed Consumer*

In this implementation we add two interceptors. We add *VectorClockInInterceptor* in the *In Chain* to update/create the *vector clock* value and store it in the database. In the *Out Chain* we added the *VectorClockOutInterceptor* to attach the *vector clock* to the outgoing message, or to update and store the *vector clock* in the case of invocations to external services.

Those interceptors can be added dynamically to the chain of interceptors. This feature allows us to use our approach without re-deploying the system under analysis.

## VII. Experiment

To investigate the impact of our approach on the service response time we designed and executed two experiments. The response time of a system can increase because the approach introduces two variables. First, we introduced two new filters in the *Pipes and Filters* pattern and the Apache CXF runtime is loaded with additional message processing tasks. Secondly, we introduced a new header element in the SOAP messages to attach the *vector clock* which increases the size of the messages passed between services.

We performed two experiments in which we measure the impact of the instrumented Apache CXF framework (*Experiment 1*) and the impact of the increasing size of the messages (*Experiment 2*) on the response time.

To perform our experiments the Apache CXF framework 2.4.1 is instrumented as described in the previous section. Tomcat 7.0.19 is used as an application server and Hibernate 3 as Java persistence framework. On the hardware part two platforms are connected through a 100 Mbit/s Ethernet connection:

- **Platform 1**: MacBook pro 6.2 , processor 2.66 GHz Intel Core i7, memory 4 GB DDR3, Mac OS 10.6.5.
- **Platform 2**: MacBook pro 7.1 , processor 2.4 GHz Intel Core 2 Duo, memory 4 GB DDR3, Mac OS 10.6.4.

Each platform uses a MySQL 5.1.53 (Community Edition) database to store the *vector clocks* values for subsequent dependencies extraction. Execution times are measured using the Java timer method, *System.currentTime()*. This method returns the current value of the most precise available system timer, in milliseconds (ms).

### A. Experiment 1

In the first experiment we investigate the impact of the instrumented version of the Apache CXF framework on the response time. We implemented the example shown in Figure 2 deploying the services within the boundary on *Platform 1* and the external service on *Platform 2*. We deployed the services within the system in one platform to achieve more accurate timing and eliminate the network overhead, which is not relevant for this experiment. Moreover the implementation of each service contains only the logic needed to invoke other services. We measured the response time of the service *OpenAccount* in three different scenarios:

- **NoClock**: we executed the system without our *vector clock* approach.
- **Clock**: we executed the system with our *vector clock* approach.
- **ClockNoDB**: we executed the system with our *vector clock* approach without storing the *vector clocks* values in the database.

For each scenario we executed the system 1000 times to minimize the influence of the operating system activities. Figure 5 shows the box plots of the response time measured for the three different scenarios while the following table shows median and average values in milliseconds.

| Scenario | Median (ms) | Average (ms) |
|---|---|---|
| NoClock | 116.6 | 108 |
| ClockNoDB | 249.4 | 226 |
| Clock | 286.4 | 275 |

The results show that on average the difference among the response time is 167 ms between the scenarios with and without *vector clocks*. The overhead due to the storage in the database using Hibernate 3 is on average 49 ms.

The difference measured is relevant, but it is relative to a system which involves the execution of 7 services without any business logic. The impact of our approach can be lower

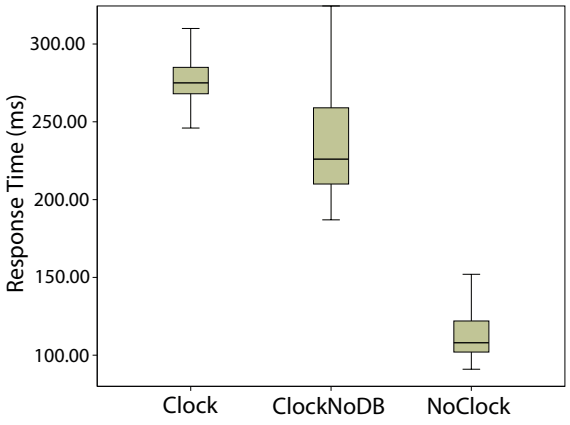


Figure 5. Box plots of the response time in milliseconds obtained for the Experiment 1

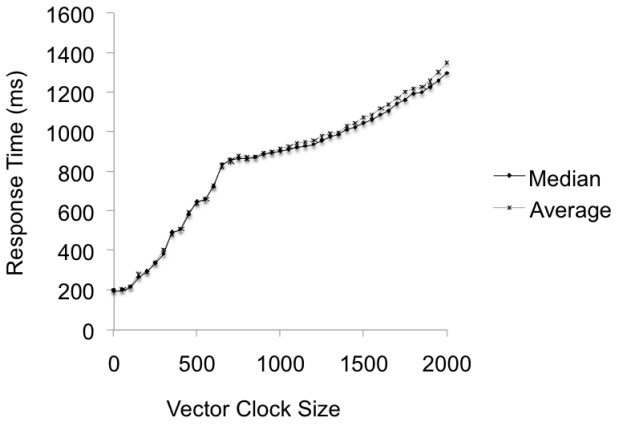


Figure 7. Average and median response time in milliseconds when increasing the *vector clock* size for experiment 2

in real systems since the increase in milliseconds introduced by the instrumented Apache CXF framework is expected to be a small percentage of the total response time when additional logic is also executed.

### B. Experiment 2

In the second experiment we investigate the impact of the increasing message size on the response time. We implemented the system shown in Figure 6. The system is composed of 12 web services that we labeled from 1 to 12. Each web service $Service_i$ invokes the $Service_{i+1}$, except the last service $Service_{12}$. The invocations among services are synchronous. To take into account the network overhead we deployed the $Service_i$ on the *Platform 1* if $i$ is an odd number and on *Platform 2* if it is even. Similarly to *Experiment 1* the services' implementations do not contain any business logic except the logic needed to invoke the other service.

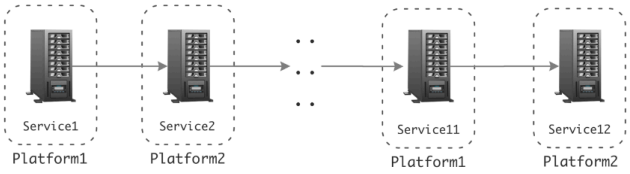


Figure 6. System deployed to perform the Experiment 2

We measure the response time of the service *Service1* while increasing the *vector clock* size from 1 to 2000 pairs. The *vector clock* is added to the message sent to *Service1* that forwards the message to *Service2* until the last service of the execution flow is reached. For each *vector clock* size, this scenario is executed 1000 times to minimize the influence of the operating system activities. The *vector clocks* are not stored to the database in order to achieve more accurate time measures.

Figure 7 shows the median and average of the measured response times for each *vector clock* size. As shown by the plot the increasing size of the messages has a relevant impact on the response time. Basically, the more unique services are invoked along the execution flow the higher the response time.

### C. Summary of the results

Our experiments measured the impact of the approach on the response time. This impact is mainly due to the increasing size of the SOAP messages. The instrumentation of the CXF framework can be a minor issue for real systems.

In order to validate whether the increase in message-size is not problematic in practice, we counted the number of services and operations in a set of industrial systems which use web services. These industrial systems have been previously analyzed by the Software Improvement Group[11] and cover a wide range of domains. The following table reports the frequencies of the number of operations and the number of services within these systems:

| #Services | #Systems | #Operations | #Systems |
|---|---|---|---|
| 1-10 | 31 | 1-10 | 13 |
| 11-100 | 6 | 11-100 | 17 |
| 101-201 | 4 | 101-500 | 9 |
| | | $> 501$ | 2 |

According to these results, applying our approach to extract dependencies in the biggest system (composed of 201 services) in our repository would lead to an increase of the response time of 140 ms in the worst case. This difference is significant for a system without any business logic, but we believe it is only a small percentage of the response time in real systems. In our future work we plan to investigate the impact of our approach in a subset of those systems.

[11] http://www.sig.eu

## VIII. Conclusion & Future Work

In this paper, we presented a novel approach to extract dynamic dependencies among services using the concept of *vector clocks*. They allow the reconstruction of an accurate dynamic dependency graph from the execution of a service oriented system.

We implemented our approach into the Apache CXF framework using the *Pipes and Filters* pattern. This pattern makes our approach portable to a wide range of SOA platforms, such as *Mule ESB* and *Apache Axis2*.

The information retrievable with our approach is of great interest for both researchers and developers of service-oriented systems. Amongst others, the dependencies can be used to study service usage patterns and anti-patterns. In addition, the information can be used to identify the potential consequences of a change or a failure in a service, also known in literature as *change* and *failure impact analysis*.

As future work, we plan to apply our approach to extract dependencies in both open-source and industrial systems. The extracted graphs allows us to measure important quality attributes of the systems under analysis, such as *changeability*, *maintainability* and *analyzability*.

Moreover, we plan to further investigate the impact of our approach on the response time of industrial systems. If the impact is significant, we plan to improve our approach to minimize the introduced overhead.

## Acknowledgment

This work has been partially funded by the NWO-Jacquard program within the ReSOS project.

## 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] S. A. Bohner. Software change impacts - an evolving perspective. In *ICSM*, pages 263–272, 2002.

[3] 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.

[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. Perepletchikov and C. Ryan. The impact of service cohesion on the analyzability of service-oriented software. *IEEE T. on Software Engineering*, 37(4):449–465, 2011.

[10] M. Perepletchikov, C. Ryan, and K. Frampton. Towards the definition and validation of coupling metrics for predicting maintainability in service-oriented designs. In *OTM Workshops (1)*, pages 34–35, 2006.

[11] M. Perepletchikov, C. Ryan, and K. Frampton. Cohesion metrics for predicting maintainability of service-oriented software. In *QSIC*, pages 328–335, 2007.

[12] M. Perepletchikov, C. Ryan, and Z. Tari. The impact of service cohesion on the analyzability of service-oriented software. *IEEE T. Services Computing*, 3(2):89–103, 2010.

[13] D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *Proceedings of the 2011 27th International Conference on Software Maintenance [To appear]*, 2011.

[14] D. Romano and M. Pinzger. Using vector clocks to monitor dependencies among services at runtime. In *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications*, QASBA '11, pages 1–4, 2011.

[15] R. A. Shawn A. Bohner. Software change impact analysis. *IEEE Computer Society Press*, 1996.

[16] S. Wang and M. A. M. Capretz. A dependency impact analysis model for web services evolution. In *Proceedings of the 2009 IEEE International Conference on Web Services*, ICWS '09, pages 359–365, Washington, DC, USA, 2009. IEEE Computer Society.

TUD-SERG-2011-034
ISSN 1872-5392