*2012 International Conference on Network and Computational Intelligence (ICNCI 2012)*
***IPCSIT vol. 46 (2012) © (2012) IACSIT Press, Singapore***

# SOAP Handlers in Prediction of Byzantine Web Services

Sankaranarayanan Murugan[1+] and Veilumuthu Ramachandran[2]

[1] Research Scholar, Faculty of Computer Science and Engineering, Sathyabama University, India.
[2] Professor, Department of Information Science and Technology, Anna University, India.

**Abstract.** The main objective of this paper is to detect the presence of suspicious code in the Web services deployment engine using SOAP handlers. The Web services deployed in application servers are infused with malicious codes intentionally to make the service behave in an abnormal manner and thereby generating faulty response to the client. This kind of Byzantine behaviour is achieved effortlessly by incorporating aspects into service methods without the knowledge of the service provider. In the proposed work, SOAP handlers are introduced into the Web services deployment engine for intercepting the clients request message in order to detect the presence of malevolent Classes. The directory structure of the deployed service is investigated based on the information provided in the service descriptor file. The service provider is informed about the presence of suspicious code (if any) and the service is deactivated until the service owner validates the reported message. Incorporating this methodology into application servers shall increase the trust level of the services and the clients are assured with fault free service.

**Keywords:** Aspect, Byzantine, Message Context, SOAP Handler

## 1. Introduction

Web services have become the de-facto Internet based heterogeneous technology for computing distributed business transactions. Identifying and elimination of faults in the complex business system is a major challenging task. Sometimes the identified faults behave abnormally by exhibiting Byzantine behaviour, which may go undetected often, as it continues to work and produces results which are illegitimate that cause business loss, customer dissatisfaction, loss of reputation and various other factors. This kind of uncharacteristic behaviour of faults is referred to as Byzantine fault or Byzantine behaviour which is more common in this age of Internet, where systems are injected with faults that are very difficult to identify, locate and eliminate.

The Byzantine Generals problem [1] is built around an imaginary General in defense who makes a decision to attack or retreat, and must communicate the decision to his lieutenants. The general and some of the lieutenants may be traitors. Traitors cannot be relied for proper communication of orders; worse yet, they may actively alter messages in an attempt to subvert the process. The generals are collectively known as *processes.* The general who initiates the order is the *source process.* The orders sent to the other processes are *messages.* The general and lieutenants those send faulty messages are traitorous and termed as *faulty processes*. Loyal general and loyal lieutenants are *correct processes.* The order to retreat or attack is a message with a single bit of information: 1 or 0. Lamport et al [1] proposed an algorithm to eliminate the Byzantine fault in which an agreement is arrived based on the messages that are exchanged among the processes.

To enhance the reliability of the Web services, it is not only necessary to handle the crash faults i.e. physical faults but also efforts should be taken to monitor and to handle the Byzantine faults due to the

[+] Corresponding author. Tel.: + 919940217379
*E-mail address*:snmurugan@live.com

untrusted communication environment in which they operate. Wenbing Zhao [2] had developed a Byzantine Fault Tolerance framework for Web services, which operates on top of the standard SOAP messaging framework with minimum changes in the Web applications. The Byzantine Fault Tolerance framework is implemented as a pluggable module and hence this model also supports inclusion of new fault tolerance requirements.

In the proposed model, faults are infused into Web services using aspects to exhibit Byzantine behaviour. Aspect Oriented Programming (AOP) is an extension of meta-programming that offers a provision for handling cross-cutting concerns that can be plugged into any of the widely adopted programming languages. AOP improves the software quality by reducing code tangling and separating the concerns. The fault generating code is created using aspects explicitly and it is separated from the actual implementation of the business logic. The aspect's advices do not require any explicit invocation as they get triggered along with the service.

Zhang et al [3] have developed an exception softening methodology to handle the exception faults effectively in AspectJ environment. They have analyzed and summarized several exception fault types of AspectJ and illustrated the way with appropriate examples to analyze the impact of exception faults on program control flow. Sevilla et al [4] envisaged the role of Aspect Oriented Programming in distributed component services with respect to distribution, fault tolerance and load balancing. Usually the code for providing QoS parameters (both functional and non-functional) is merged with the business logic and hence it is harder to develop, maintain and reuse the code. In the proposed aspects based model for Byzantine agreement, the Byzantine behaviour identification module is completely decoupled from the Web service, which is meant for its intended task. Domokos et al [5] have modelled the fault tolerant structures using aspects and this framework is extended for automatic construction of an analysis model, which is a dependability model that is used to determine the non-functional properties of the system. In order to improve the reliability and availability of distributed object oriented systems, Herrero et al [6] have introduced object replication mechanisms and presented a replication model, JReplica, which is a Java fault tolerance language based on Aspect Oriented Programming. This replication model separates the specification of the replication code from the functional behaviour of objects by providing a high degree of transparency. JReplica provides facilities to the programmers to introduce new behaviors for specifying different fault tolerant requirements. To enhance the reliability of the Web services, it is not only necessary to handle the crash faults but also efforts should be taken to monitor and to handle the Byzantine faults due to the untrusted communication environment in which they operate.

Many aspect oriented application programming interfaces are available as open source for different programming paradigms. In the proposed model, aspects are created using Java based AspectJ [7] to inject faults into the Web service. In AOP, *Joinpoints* are well defined check points in the flow of the application, which may be (i) method call or return, (ii) bean operations (set and get) and (iii) exception handler entry point. A collection of joinpoints is termed as *pointcuts*. *Advices* are codes that will execute on some conditions like *before*, *after* or *around* the joinpoint. Aspect is like a class which includes *pointcuts* and *advices* for implementing the *cross-cutting concerns*. *Concern* refers to a specific purpose i.e. a portion of code for which the aspect is introduced. *Weave r* combines the classes and aspects for constructing the actual application.

Handlers are pluggable components for intercepting the SOAP messages either on the client or server end [8]. Handlers are stateless and invoke any intermediate business processing logic by considering the input from the MessageContext. Handler provides user defined functionality such as reliability, security and various other quality issues. A MessageContext is a collection of properties to hold the metadata during message exchange and is available to the handlers. In the proposed approach handlers are coded for the server end and input is the SOAP request payload received from the client i.e., incoming MessageContext. Separate inbound and outbound MessageContext properties are available for SOAP Header, SOAP Body and SOAP attachments. A "phase" is defined as a logical collection of handlers. An AXIS engine invokes phase in a given flow and then the phase will sequentially invoke all the handlers in it. Phase can be "global" or "operation". A global phase is invoked for any deployed service and operation phase is applied for only a specific operation(s). Phase rule has the properties like phase name, first handler of the phase, last handler of

the phase, the placement of handler to a phase can be at “before” or “after” or “before and after”. A flow is a collection of phases and can be of “InFlow” (request message), “OutFlow” (response message), “InFaultFlow” (faulty incoming request) and “OutFaultFlow” (faulty outgoing response).

## 2. Detecting Suspicious Code using SOAP Handlers

SOAP handlers are adopted to detect the presence of suspicious code and this approach provides a methodology to prevent the services from behaving abnormally and hence Byzantine faults may be averted. In the real time scenario many composite Web services are involved while processing the client’s request. The response to the client is based on the individual response as provided by each intermediary service. If any one of the service provide a negative response by denying the request, as per the distributed environment paradigm all the responses of the other services are discarded and an exception message is sent as a response to the client. The service that behaves differently from other services may not be a genuine one i.e. the service is exhibiting Byzantine behaviour and it is an untrusted one. If the service infected with suspicious code is a part of the composite Web service, then the service exhibits malicious behaviour and dispatches faulty response to the coordinating service. This response could lead to failure of the entire service when goes undetected. The technique proposed in this paper does not affect the performance of the service as the handlers are executed inherently. The algorithm to detect the malicious code is given below:

### 3.1. Algorithm

1. Intercept the SOAP Request message sent by the client. Obtain the name of the service method that is to be invoked.
2. Analyze the service descriptor file “services.xml”
   a. if <serviceGroup> element is present then parse the list of <service> elements. Obtain the service class details (package and Class name) by manipulating the element <parameter name="ServiceClass">
   b. otherwise retrieve the value of the “parameter” a child element of “service” element.
3. Navigate to the “classes” directory where the service has been deployed based on the package name retrieved from step 2.
4. Search for the classes that are not specified in the “service group” or “service” element based on the “classes” retrieved in step 2.
   a. If any other Class is present, analyze the Class for the presence of suspicious code.
      i. In the Class, search for the method (as obtained in step 1). If the search string is present then the service is bound to be a suspicious one and it may generate faulty response.
      ii. The information about the presence of additional Class is intimated to the service provider.
      iii. The response to the client is blocked and an exception response is dispatched.
      iv. The service is blocked until the service provider acknowledges the message or rectifies the infected service.
   b. Otherwise, the application server continues with processing the clients request and dispatches the appropriate response to the client.

### 3.2. Implementation

The proposed model is implemented in AXIS2 framework for Web services deployed in Tomcat Web container. An aspect code is introduced to inject faults into the Web service. To detect the injected aspect code into the service, SOAP handlers are implemented as AXIS2 modules.

#### 3.2.1 Injecting Faulty Code using Aspects

This section explains how a service can be associated with a faulty code using aspects. The class WebApplicationService is a collection of Web methods (each method represents a service).

public class *WebApplicationService* {
public <*return type*> web_Method1(*arg1, arg2, ..., argn*) { /* *business logic* */ }
….
public <*return type*> web_Methodn(*arg1, arg2, ..., argn*) { /* *business logic* */ } }

The class WebApplicationService may be associated with an aspect code that executes along with the service without its knowledge. This is achieved by:

```
public aspect WebApplicationAspect {
 public pointcut aspect_Method_Call1(a1, a2, ..., an): execution (public <return type>
                    WebApplicationService.web_Method1(arg1, arg2, ..., argn )) && args(a1,a2, ..., an);
 <return type> around(int a, int b): aspect_Method_Call1(a1, a2, ..., an) {
    /* Modify the actual business logic with malicious code */
    return value;  }
…
 public pointcut aspect_Method_Calln(a1, a2, ..., an): execution (public <return type>
                    WebApplicationService.web_Methodn(arg1, arg2, ..., argn )) && args(a1,a2, ..., an);
 <return type> around(int a, int b): aspect_Method_Calln(a1, a2, ..., an) {
    return value;  } }
```

In the above code fragment, the pointcut "aspect_Method_Call1" gets triggered whenever the service method "web_Method1" is invoked by the client. Since an "around" aspect is associated with the method, the business logic present in the aspect is executed and the response of this aspect is returned to the client and not the result that is processed by the actual service. This execution take place without the knowledge of the service provider as there is no external invocation is required. The aspect code returns a faulty response to the client. Similar is the case with other service method, in which the aspect returns the irrelevant output. In a similar manner, basic functionality of the complex business algorithms can be modified and make the system to behave in a strange manner. This type of malicious code is detected by incorporating the proposed methodology implemented using SOAP handlers and deployed as an AXIS2 module.

A user defined SOAP Handler is created by extending the abstract class "AbstractHandler" which implements the interface "Handler". An "InvocationResponse" is an inner class of the interface Handler which encapsulates the method "invoke" will be called on each registered handler when a message needs to be processed. The parameter MessageContext of invoke method is capable of retrieving the type of message to be processed (inbound or outbound), message flow, operation and message exchange pattern. The method returns the next step in the message processing flow. The return value can be of "CONTINUE" (message can be forwarded to the next level of processing), "ABORT" (terminates the operation and will not proceed further) and "SUSPEND" (when some of the required conditions are not met and further processing is not allowed). AxisFault is an exception, a base class for exceptions that are mapped to SOAP faults which consists of a fault string, fault code, fault actor and fault details.

```
public class ServiceAspectHandler extends AbstractHandler{
    public InvocationResponse invoke(MessageContext msgContext) throws AxisFault {
    // Algorithm to determine the presence of suspicious code
            if (suspicousCode==true){
                    // Intimate the information to the Service Provider by throwing an AxisFault.
                    return InvocationResponse.SUSPEND; }
            else
                    return InvocationResponse.CONTINUE; } }
```

Lifecycle methods of interface Module is implemented in "HandlerModule.java" which includes "init", "shutdown", "engageNotify", "getPolicyNamespaces", "applyPolicy" and "canSupportAssertion". The first two methods are used to control the module at the time of initialization and termination. The method "engageNotify" may be involved in validating the module, adding a policy and disengage the module, "applyPolicy" evaluates specified policy for the currently processing message and "canSupportAssertion" method evaluates to true when the module supports assertion.

```
public class HandlerModule implements Module {
public void init(….)
public void engageNotify(…)
public void shutdown(…)
public void applyPolicy(…)
public boolean canSupportAssertion(…)                }
```

Module descriptor file is specified in “module.xml”. InFlow is configured with user defined “ServiceAspectHandler” and the phase handler is assigned with the value “Transport”.

```
<?xml version="1.0" encoding="UTF-8"?>
<module name="aspectmodule">
<InFlow>
<handler name="ServiceAspectHandler" class="aspect.handler.ServiceAspectHandler"/>
<order phase="Transport" after="RequestURIBasedDispatcher"
before="SOAPActionBasedDispatcher"/>
</InFlow> </module>
```

Incorporate the handler module into the service descriptor file “services.xml” by using the element <module>. Hence SOAP handlers will be always available whenever the service is invoked.

```
<?xml version="1.0" encoding="UTF-8"?>
<service name="webcservice" scope="application">
<messageReceivers>
<messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out" class="RPCMessageReceiver"/>
</messageReceivers>
<parameter name="ServiceClass">WebApplicationService</parameter>
<module ref="aspectmodule"/>
</service>
```

A generic handler to detect suspicious code for all the services deployed in the Web services engine is incorporated in the “axis2.xml” as:

```
<axisconfig>
…
<module ref="aspectmodule"/>
</axisconfig>
```

It is observed that the average round trip time (RTT) and throughput does not show any significant difference when SOAP handlers are introduced for detecting the suspicious code and hence performance issues are not of major concern.

## 3. Conclusion

An effective mechanism is developed to detect the presence of malicious code in the Web application servers using SOAP Handlers. To tolerate the Byzantine behavior of certain services and to take appropriate decisions where the services are exhibiting traitorous response in a distributed environment, the existing SOAP communication framework associated with AXIS2 runtime is extended using SOAP handlers without detrimental to the normal responsibilities. Once the service is identified as suspected, further requests from the client are blocked until the service provider clears the fault. If the blocked service is a traitorous one, then the service provider has to rectify the faulty nature of the service and redeploy into the Web application server. The developed model is a pluggable module and hence it can be easily attached or detached at any point of time. The developed methodology can be attached with individual services as an add-on feature based on request and the same can be extended to detect the presence of malicious code in the Web service deployment engine globally.

## 4. References


[1] Leslie Lamport, Marshall Pease and Robert Shostak. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*.1982 4(3):382-401.

[2] Wenbing Zhao. BFT-WS: A Byzantine Fault Tolerance Framework for Web Services. in *Proc. Of 11th International IEEE Conference Enterprise Distributed Object Computing Conference Workshops.* USA. 2007. pp. 89-96.

[3] Zhang Ji-de and Yao Ying. Analysis of exception fault types based on AspectJ. in *Proc of IEEE ICCASM*. China. 2010. Vol. I, pp. 287-289.

[4] Diego Sevilla, Jose M. Garcia, Antonio Gomez. Aspect-Oriented Programming Techniques to support Distribution, Fault Tolerance, and Load Balancing in the CORBA-LC Component Model. 2007. In *Proc of IEEE NCA*. pp. 195-204.

[5] P. Domokos and I. Majzik. Design and Analysis of Fault Tolerant Architectures by Model Weaving. 2005. In *Proc HASE*. Germany. pp. 15-24. doi:10.1109/HASE.2005.8.

[6] J. Herrero, F. Sanchez, and M. Toro. Fault Tolerance AOP Approach. 2001. in *Proc of International Workshop on Aspect-Oriented Programming and Separation of Concerns.* Lancaster University, UK. pp.44-52.

[7] AspectJ, *http://eclipse.org/aspectj*

[8] Apache. AXIS2, *http://axis.apache.org/*