Modern Applied Science; Vol. 9, No. 9; 2015
ISSN 1913-1844 E-ISSN 1913-1852
Published by Canadian Center of Science and Education

Detection and Elimination of Byzantine Faults Using SOAP Handlers in Web Environment

Sankaranarayanan Murugan[1] & B. Muthukumar[1]

[1] Professor, Faculty of Computing, Sathyabama University, Chennai, Tamilnadu, India

Correspondence: Sankaranarayanan Murugan, Department of Information Technology, Faculty of Computing, Sathyabama University, Jeppiaar Nagar, Old Mahabalipuram Road, Chennai-600119, Tamilnadu, India. Tel: 91-994-021-7379. E-mail: snmurugan@live.com

Received: February 27, 2015 Accepted: March 20, 2015 Online Published: August 30, 2015

doi:10.5539/mas.v9n9p161 URL: http://dx.doi.org/10.5539/mas.v9n9p161

Abstract

Detection and elimination of Byzantine faults in the Web services environment by applying the features of SOAP handlers is the principle objective of this work. The Web services may sometimes be infused with suspicious modules intentionally for them to behave in an abnormal manner. By introducing faulty aspects into the service deployment on the fly makes the Web service to generate Byzantine faults. The application servers too shall be infected with malicious aspect codes for generating Byzantine faults and therefore all the deployed services may be-come victim for dispatching erroneous response. In the proposed work, SOAP handlers are induced into the Web application server for manipulating the clients request message for detecting the occurence of Classes that behave in an abnormal manner in both the application servers and in service deployments. The service that is being affected by the presence of malicious classes is deactivated, till the faulty service is replaced or repaired by the original service provider. When these type of SOAP handlers are induced into the Web application servers, the reliability of the service is increased and thereby the clients are guaranteed with error free response. It is also observed that when SOAP handlers are introduced in detecting Byzantine faults, there was no performance degradation either in server level or at service level.

Keywords: aspect, AXIS2, byzantine, SOAP, web service

1. Introduction

Web services have become the de-facto Internet based heterogeneous technology for computing distributed business transactions. Faults in distributed systems arise due to congestion in networks, network link failures, intermediate component faults, server overload, hardware or software faults at both client and server ends, denial of service attacks and various other factors. Resource consumption, fault handling procedures and the strength of the fault model are the certain constraints involved in fault tolerant framework, which leads to performance degradation. The resources that are required while processing the fault handling mechanism are CPU, memory, I/O, bandwidth usage etc. The performance of the service may be degraded due to fault detection latency, replica launch latency, fault recovery latency etc. The fault model is measured by client transparency, group communication, granularity in fault detection and recovery etc. To improve the scalability of the servers under heavy load, clustering techniques are applied. A cluster is a set of networked servers that offer a single system image and provides additional processing capabilities. Clustering techniques adopt redundancy approach to achieve scalability and high availability of distributed services. The clusters create fault recovery algorithms for achieving not a single point of failure either at the entry level or during the clustering process.

Detection and elimination of malicious faults in the complex business services is a major challenging task. The services that exhibit Byzantine behaviour may get undetected as the system continues to produce the results which are faulty, incorrect and manipulated data. These illegitimate results may lead to business loss, customer dissatisfaction, loss of reputation and several other reliability issues. The uncharacteristic behaviour of the service response are often referred to as Byzantine Fault which is most of a common issue in today world of the Internet Technology where the faults that are injected into the services which are very difficult to identify, locate and eliminate.

Byzantine Generals Problem (Lamport et al., 1982) is an imaginary problem, in which the General in defense have to make a decision for attacking or retreating and inform the message as the decision to all the lieutenants

under his control. There are chances that some of the Generals and lieutenants' are themselves traitors. Durign the transfer of communication between the lieutenants and the Generals, it is the traitorous ones who may send faulty messages to every other army personnel's which leads to loss of war. In the system of Web, the generals may be collectively referred to as the processes. The General that sends order to every other lieutenant and generals is termed as Source Process. Messages here are referred to as the orders. Faulty processes are known as traitors. Correct processes only share the actual information among their peers. In order provide a valid transaction and conclusion about the decision to accept or reject a message, a bit of data either 1 or 0 is applied. A solution to the problem based on the exchange of messages is formulated by Lamport et al., (1982) for eliminating the Byzantine fault through an agreement. For providing efficient reliability of the Web environment, in an untrusted environment the system should not consider only the physical or crash faults but also the malicious behaviour that is being exhibited without the knowledge of the service provider. Wenbing Zhao (2007) proposed a framework for handling Byzantine faults which works on top of the SOAP messages. In the Web application server, a minor modification is to be done for detecting and handling the Byzantine faults. In this framework the fault tolerance system is included in the Web service deployment engine as a pluggable module. In the proposed work, for exhibiting malicious behaviour the faults are injected into Web services using the concept of meta-programming. 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 quality of the software by reducing code tangling and separating the concerns. The original implementation of the business logic and the fault producing code are separated with the help of aspects. The fault generating code does not require any external invocation as it gets triggered along with the original service and responses with the faulty data as it is being coded with the help of aspects.

An efficient method for handling exception faults using AspectJ environment is developed by Zhang and Yao (2010). In this approach, exceptional faults of several types are analyzed and are summarized. They also have illustrated the impact of exception faults on the flow of the program with several case studies. The environment in which all the examples are generated is by using AspectJ API. For effective prediction of load balancing, fault tolerance and distribution of services in the component based distributed services environment using aspect oriented programming is proposed by Sevilla et al.,(2007). As per this approach, the logic for the functional and non-functional Quality of Service parameters is separated from the implementation of the actual business logic. Since the QoS parameters and business logic are separated very minimal effort is to be taken for developing, maintenance and reusage of the code. In the proposed work for handling Byzantine faults using the SOAP handlers the malicious execution behaviour is decoupled from the core business service. For improving the reliability and availability of the distributed object oriented system. Domokos and Majzik (2005) have developed a framework in which an aspect based fault tolerant structure has been developed. In this work, an automatic construction of an analysis model for determining the non-functional properties of the system is proposed. A replication model JReplica, a java based fault tolerance language using Aspect Oriented Programming is proposed by Herero et al., (2006) which separates the specification of the replica code from the functional behaviour of the objects. A high level degree of transparency is provided in this approach. Several fault tolerant requirements have been introduced using JReplica based on the nature of the fault.

In a system of several coordinating Web services, there is a possibility that one or more services may behave maliciously by exhibiting Byzantine behaviour and it is a challenging task to identify and eliminate them (Murugan and Ramachandran, 2012). As per this approach, the response from all the participating services is captured and are analysed to identify the presence of Byzantine faults before dispatching the response to the client and it is claimed that the client will receive a fault free response. Using aspect oriented programming approach, decision making in the presence of Byzantine faults is arrived by capturing the responses of before, after and during the execution of the service (AspectJ, 2013). The model also identifies the origin of the fault and provides a mechanism to detect and eliminate the Byzantine faults. 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 (AspectJ, 2013) 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. Weaver combines the classes and aspects for constructing the actual application. AspectJ, AspectWerkz, Nanning, Prose (PROgrammable Service Extensions) are some of the tools available for implementing Aspect oriented programming in various programming languages. Aspects do have several advantages but it lacks in supporting maintenance and

management, hard to debug and very few tools are available to implement. These aspect concepts are applied for making the system to behave in an abnormal manner, to identify the faulty services and application servers and to transmit the genuine response to the client.

In SOAP messages, the pluggable components referred to as Handlers is included into the SOAP messages for intercepting the messages that are transmitted between the sender and receiver. These handlers can be introduced either at the client end or on the server end (Apache AXIS2, 2013). Using the input from the class MessageContext any intermediate execution of the business logic can be captured or invoked with the help of SOAP Handlers. SOAP Handlers by default they are stateless by default. Handler provides user defined functionality such as reliability, security and various other quality issues. In the proposed approach, an incoming MessageContext is the request payload from the service requester end which is then transferred to the service provider. To provide and store the metadata during the message exchange for handlers, a collection of properties MessageContext is used. For incoming and outgoing messages, a unique inbound and outbound message exchange properties is provided by the class MessageContext is available for the SOAP message components Header, body and attachments. Logical collection of SOAP handlers are referred to as a “phase” and they can be either available as “global” or “operation”. The phase will invoke all the associated handlers when the AXIS engine triggers a phase in a given flow (inbound or outbound). The global phase is triggered for any of the deployed service in the Web deployment environment whereas the operation phase is only for a specific service. Phase name, first hander and last handler are the phase rules and the handlers can be made available at either “before” or “after” or “before and after”. The phase collection is referred to as a “flow” and it can either any one of the following (a) InFlow (request message), (b) OutFlow (response message), (c) InFaultFlow (faulty incoming request) and (d) OutFaultFlow (faulty outgoing response).

The Byzantine faults may be exhibited in Web services through several ways. The services may be infused with malicious methods, duplication of original service for generating faulty response, faulty codes may be introduced into the message receivers of Web service deployment engine and any intermediary node may generate Byzantine behaviour. It is proposed to detect and eliminate the Byzantine faults in the presence of faulty services and faulty Web service deployment engine.

2. Detection of Malicious Behaviour in a Service

To detect and eliminate the presence of malicious and suspicious code fragment, SOAP handlers are implemented and thereby preventing the abnormal behaviour of services and therefore it is possible to eliminate the possibility of Byzantine faults. Several Web services may be involved in providing a solution to the request from the client which is usually referred to as Composite Web services. The intermediary service provides the individual response to the service that is being invoked and the final processed response is dispatched to the client. In this composite process, if any of the service behaves in a faulty manner then the entire process may be reflected with the error proned messages. Under normal situation, if it is found that any of the service behaves abnormally then the entire solution is ignored and the entire process is repeated. In sometimes, if the faulty node is not identified, then the traitorous node continues to function and dispatches faulty response which goes undetected. This is leads to the total failure of the entire service network. In the proposed work, the system is capable of obtaining the error free response with the help of SOAP Handlers.

In order to detect the presence of suspicious classes in the service, the service descriptor file is analyzed by using SOAP handlers. Figure 1 shows the steps for detection of malicious code in the service level.

The flow of sequence in Figure 1 is:

1. The SOAP request message dispatched by the client is intercepted with the help of SOAP Handlers and the appropriate service to be invoked is retrieved by using the name attribute of the service.
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. Proceed to the location of the service classes where it has been deployed with the knowledge of package name (as obtained 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.

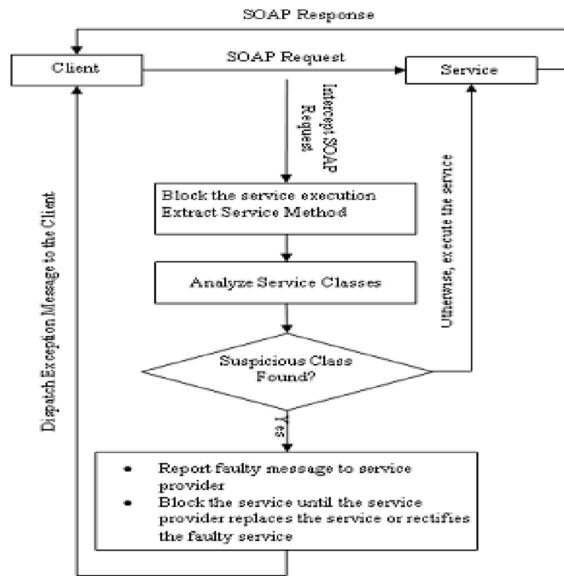

Figure 1. Detection of Suspicious Services in Web Application Services

a. If any other Class is present other the one present in the service deployment package and then look for the presence of any malicious code present.

(i) In the Class, search for the method. If the search string is present then the service is bound to be a suspicious one and it may generate faulty response.

(ii) The availability of new class is informed to the original servie provider. This class may be of malicious one.

(iii) An exception message is dispatched to the client by blocking the faulty response.

(iv) Only on rectifying the infected service or replacing it, till that time the service is blocked. Now the service provider is able to rectify the service.

b. Otherwise, the Web application server continues with processing the clients request and dispatches the fault free response to the client.

3. Detection of Byzantine Faults in Web Services Deployment Engine

In order to detect the malicious behaviour at Web service deployment engine, include the "BFTModule" as an additional add-on feature to the engine. By enabling the "BFTModule" for the service, the process of detecting Byzantine faults is applicable for the service and the same has to be included in the service descriptor file. Since the "BFTModule" is to be accessed globally, the module is to be defined in the configuration file of the Web service deployment engine. Figure 2 represents the flow of sequence service invocation by incorporating the user defined module "BFTModule".

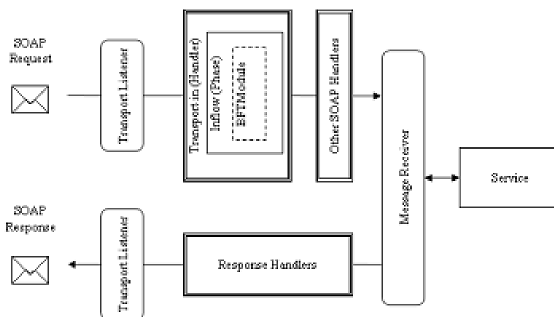

Figure 2. AXIS2 SOAP Handler with BFTModule

The BFTModule is introduced into the "Inflow" phase of the "Transport in" Handler of SOAP message. The response from the module BFTModule is responsible for the service invocation which is the original serice to be triggered and is achieved through the Message Receivers.

4. Process involved during Execution

AXIS2 framework is used as the implementation platform for detection of malicious code and is deployed into the Tomcat Web Application Service. Using the aspects, the faults are injected into the Web services. The module "BFTModule" is integrated into the Web service engine for identifying the faulty code. In the configuration of AXIS2 engine for Web services the BFTModule is integrated.

4.1 Injection of Malicious Code

The integration of faulty code in the service using aspects is discussed in this section. Consider a calculator service with basic operations addition, subtraction, multiplication and division.

```
public class CalculatorService  {
        public int add(int a, int b)      { return a+b; }
        public int sub(int a, int b)      { return a-b; }
        public int mul(int a, int b)      { return a*b; }
        public int div(int a, int b)       { return a/b; }     }
```

An aspect code may be associated with the class CalculatorService without its knowledge. The aspect gets triggered along with the original service. This is achieved by:

```
public aspect CalculatorAspect {
public pointcut subMethodCall(int a, int b): execution (public int CalculatorService.sub(int, int)) && args(a, b);
int around(int a, int b): subMethodCall(a, b)
                {      return a+b;              }
public pointcut divMethodCall(int a, int b): execution (public int CalculatorService.div(int, int)) && args(a, b);
int around(int a, int b): divMethodCall(a, b)
                {              return a*b;      }
}
```

The pointcut "subMethodCall" gets triggered implictly during the execution of the service method "sub" (when invoked by the client). In the method "subMethodCall" the advice "around" is attached, therefore whenever the service sub is executed, it is the faulty response from the "subMethodCall" is returned to the client. The response generated by the method "sub" is blocked from dispatching it to the client. The same sort of issues is present in the service method "div" will be experienced. In the "div" method for performing the division operation, the product of two numbers is returned. With a similar approach, the principle functionality of the business logic is violated and the system is made to behave in an malicious or strange manner. The malicious behaviour the services is detected and eliminated with the help of SOAP Handlers is proposed in this work.

4.2 BFT Module

The module "BFTModule" is the implementation of the Lamport's algorithm for tolerating Byzantine faults in the distributed Web environment. A set of interfaces have been created for handling Byzantine behaviour in SOAP messages. The interfaces are ServiceFactoryInterface, FaultNotifierInterface, SOAPFaultInterface, ReplicaManagerInterface and BFTInterface have been implemented.

The ServiceFactoryInterface is responsible for the creation of new service, activation or deactivation of a service and the service is undeployed. Each and every service is associated with a unique ID. Using the SOAP communication framework the services (applications) are deployed into the Web Application server. The ReplicaManagerInterface controls and monitors the replicas by inclusion and exclusion of the location of the replica servers. Similar to the service identifier the replicas are also assigned with unique identification number. A HTTP Web server is used for managing the replicas. The replications techniques vertical and horizontal are implemented with the proposed system and are tested. Only for horizontal replication, the replicaLocation() method in the interface is required. The FaultNotifierInterface identifies the misbehaving replica and error proned replica. The faulty replica is returned by the interface FaultNotifierInterface. The implementation of the Lamport's solution for the Byzantine agreement is provided in the interface BFTInterface. All the nodes are

associated with the replicas and the messages are transmitted among themselves. The messages that are broadcasted among the replicas are processed and the error proned process is identified (if any). The unique replica identifier of the faulty process is reported to the replication manager. The SOAPFaultInterface retrieves the SOAP Fault codes that are generated by the replicas.

The BFTInterface in turn defines ServiceInterface, NodeValueInterface, MapRepositoryInterface and BroadcastInterface. The ServiceInterface maintains the list of participant services. The NodeValueInterface is used for retrieving current message value that is passed from one service to another service. The MapRepositoryInterface is used for defining the services hierarchy, for identifying the path in which the message communication takes place, for specifying the number of messages exchanged between the participant services. The BroadcastInterface is applied for maintaining the number of repetitions and the message transfer details.

The above set of interfaces are bundled together and deployed as a module into the AXIS2 engine for effectively handling of Byzantine faults in the Web service environment. The module has the capability to adapt to any Web server environment with minor modifications (depends on the server configuration). The module has a flexibility that, when not required it can be disabled.

4.3 Identifying Byzantine Faults using SOAP Handlers

An abstract class “AbstractHandler” is extended for creating the user defined SOAP Handler which implements the interface “Handler”. An inner class for the interface Handler labeled as “InvocationResponse” that encapsulates the method “invoke” is triggered for each call on the registered handler during the processing of the message. To retrieve the type of message for processing i.e. whether it is inbound or outbound, flow of the message, the exchange patter of the messages and the type of operation is achieved with the help of the parameter “MessageContext” available in the method “invoke”. The invoke method returns the flow of the message processing which is the next step and the return value can be either any of the following: (i) “CONTINUE” (message can be forwarded to the next level of processing), (ii) “ABORT” (terminates the operation and will not proceed further) and (iii) “SUSPEND” (when some of the required conditions are not met and further processing is not allowed). The base class in AXIS2 for the exception handling mechanism is the class “AxisFault” is mapped to the SOAP faults. The SOAP fault consists of the elements fault string, fault code, fault actor and fault details.

```
public class ServiceAspectHandler extends AbstractHandler{
public InvocationResponse invoke(MessageContext msgContext) throws AxisFault {
    if (suspiciousCode==true)
        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” is 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="BFTModule">
    <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="calcservice" scope="application">
    <messageReceivers>
        <messageReceivermep="http://www.w3.org/2004/08/wsdl/in-out"
                    class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
    </messageReceivers>
    <parameter name="ServiceClass">calc.CalculatorService</parameter>
    <module ref="BFTModule"/>
</service>
```

In the AXIS2 configuration file "axis2.xml", to detect the suspicious code using generic handler for all the service is specified.

```
<axisconfig>
...
    <module ref="BFTModule"/>
</axisconfig>
```

In the presence of "BFTModule" it is assured that the client will not receive any faulty response from the service provider. In case if any service is behaving in an abnormal manner it is detected by the "BFTModule" and informed to the service provider by the module and hence the trust level of the service is significantly increased.

5. Performance Analysis

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. Table 1 shows the comparison between the Byzantine Fault Tolerance with and without SOAP Handlers for the parameters throughput in kbps and average round trip time measured in milliseconds.

Table 1. Performance comparison for detection of Byzantine faults using SOAP Handlers.

Performance Parameter	BFT With SOAP Handlers	BFT Without SOAP Handlers
Throughput	725.83 kbps	738.53 kbps
Average Round Trip Time	8.2841 ms	8.4412 ms

The result shows that there is very minimal difference in the execution process when the Byzantine Fault Tolerance approach is incorporated through SOAP Handlers. For reliable communication, this percentage of variation does not have much impact in the client or server side processing except in the critical applications.

6. Conclusion

To detect the presence of malevolent business logic and effective mechanism using SOAP Handlers in the Web application servers and Web services using SOAP Handlers is developed. SOAP Handlers under the AXIS2 runtime environment is responsible for arriving at a decision in the presence of traitorous services which dispatches faulty responses in the distributed environment without disturbing the original responsibilities of the

server. SOAP Handlers also identifies the class that is responsible for generating faulty messages. The suspected service is suspended by Web application server and no more requests are dispatched to the appropriate service. The service is resumed back whenever the faulty service is either replaced or further approval from the owner of the service. Based on the faulty nature of the service, the service is rectified or replaced and redeployed into the Web application server and further requests are permitted. The SOAP Handler is implemented as a pluggable module and the same can be affiliated to the service or disaffiliated any point of time from the Web application server or from the Web service. In this approach, the faulty service is updated to the service provider. Since the client is assured with fault free response, the trust level of the service is increased. The Web service gets executed only when it is free from Byzantine faults as these faults are detected well in advance. In the current scenario the Byzantine faults are detected only after executing the service. Further the work can be extended to eliminate Byzantine faults in application servers for any type of application.

References

Apache. AXIS2. (2013). Retrieved from http://axis.apache.org

Aspect, J. (2013). Retrieved from http://eclipse.org/aspectj

Diego, S., Jose, M. G., & Antonio, G. (2007)Aspect-Oriented Programming Techniques to support Distribution, Fault Tolerance, and Load Balancing in the CORBA-LC Component Model, *IEEE Proceedings of Network Computing and Applications,* 195-204. http://dx.doi.org/10.1109/NCA.2007.8

Domokos, P., & Majzik, I. (2005). Design and Analysis of Fault Tolerant Architectures by Model Weaving, *IEEE Proceedings of High Assurance Systems Engineering*, 15-24. http://dx.doi.org/10.1109/HASE.2005.8

Herrero, J., Sanchez, F., & Toro, M. (2001). Fault Tolerance AOP Approach. *Proceedings of Aspect-Oriented Programming and Separation of Concerns*, 44-52.

Leslie, L., Marshall P., & Robert S. (1982). The Byzantine Generals Problem: *ACM Transactions on Programming Languages and Systems*, *4*(3), 382-401. http://dx.doi.org/10.1145/357172.357176

Murugan, S., & Ramachandran, B. V. (2012), Fault Tolerance in SOAP Communication Services *Malaysian Journal of Computer Science*, *25*(2), 67-75.

Murugan, S., & Ramachandran, V. (2012). Aspect Oriented Decision Making Model for Byzantine Agreement, *Journal of Computer Science*, *8*(3), 382-388. http://dx.doi.org /10.3844/jcssp.2012.382.388

Wenbing, Zh. (2007). BFT-WS: A Byzantine Fault Tolerance Framework for Web Services, *IEEE Proceedings of Enterprise Distributed Object Computing Conference Workshop,* 89-96. http://dx.doi.org/10.1109/EDOCW.2007.6

Zhang, Ji-de., & Yao, Y. (2010). Analysis of exception fault types based on AspectJ, *IEEE Proceedings of Computer Applications and System Modeling*, 287-289. http://dx.doi.org/10.1109/ICCASM.2010.5619408

Copyrights

Copyright for this article is retained by the author(s), with first publication rights granted to the journal.

This is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/).