BYZANTINE FAULT TOLERANCE IN SOAP COMMUNICATION SERVICES

***Murugan Sankaranarayanan*[1], *Ramachandran Veilumuthu*[2]**
[1]Research Scholar, Faculty of Computer Science and Engineering, Sathyabama University,
Jeppiaar Nagar, Rajiv Gandhi Salai, Chennai – 600 119, Tamilnadu, India. Email: snmurugan@live.com
[2]Professor, Department of Information Science and Technology, Anna University,
College of Engineering, Guindy, Chennai – 600 025, Tamilnadu, India. Email: rama@annuniv.edu

ABSTRACT

An innovative SOAP communication model with modified message exchange pattern is proposed to identify and eliminate Byzantine faults in distributed services environment. When a request is made by a client to a Web enabled application, it is usually processed by a set of composite Web services. 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. In the proposed model, the Lamport's algorithm is utilized to identify the Byzantine faults in the distributed services environment. The services may behave abnormally and exhibit Byzantine behaviour that will lead to faulty response. The response from all the coordinating distributed services is captured and analyzed to determine the presence of Byzantine faults before dispatching the actual response to the client. To induce faults into the services artificially during execution, around advice aspects are adopted. When the fault identified is legitimate, response without any modification is dispatched to the client; otherwise a revised response is generated. In the presence of traitorous behaviour, the faulty service is handled within the new message receiver named as BFTMessageReceiver which is included in the existing message exchange pattern. The BFTMessageReceiver is extended from the original message receiver used in the SOAP communication framework, and hence, the regular response dispatching capabilities are retained.

Keywords:* *Aspect Oriented Programming, Byzantine Agreement, Crosscut, Joinpoints, SOAP, Weaver, Web services

1.0 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 are often undetected, as it continues to work and produces results that are illegitimate, causing business loss, customer dissatisfaction, loss of reputation and various other factors. This uncharacteristic behaviour of faults is referred to as Byzantine fault or Byzantine behaviour, 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 reached based on the messages that are exchanged among the processes. Lamport's algorithm detects the presence of Byzantine faults for $n > 3f + 1$ where 'n' is the number of processes and 'f' is the upper bound on the number of faulty Byzantine processes. Bazzi and Herlihy [8] suggest an

alternate broadcast algorithm that tolerates any number of Byzantine processes. The consensus and broadcast can be solved if $n \geq 2F + 3f + 1$, where n is the total number of processes, f is the number of undetected Byzantine processes (permanently) and F is the number of Byzantine processes that are detected.

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 Byzantine faults due to the untrusted communication environment in which they operate. Zhao [2] 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.

Zhang et al [3] developed an exception softening methodology to handle the exception faults effectively in AspectJ environment. They analyzed and summarized several exception fault types of AspectJ and illustrated the way to analyze the impact of exception faults on program control flow with appropriate examples. Sevilla et al [4] envisaged the role of Aspect Oriented Programming in distributed component services with respect to distribution, fault tolerance and load balancing. In order to improve the reliability and availability of distributed object oriented systems, Herrero et al [5] 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.

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 are triggered along with the service.

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 [6] 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 that 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.

2. 0 BYZANTINE FAULT TOLERANCE IN DISTRIBUTED SERVICES

It is proposed to enhance the SOAP communication framework with inherent Byzantine fault tolerance mechanism. 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 services provides 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 untrusted. The existing message receiver in the SOAP communication framework is extended to handle this kind of traitorous behaviour.

Each service participating in processing the client's request sends a response to the primary or controlling service. Based on the set of responses, the primary service has to send an appropriate response to the client. Under usual conditions the primary service does not validate the messages that are received from other services. In the proposed model, the set of responses received by the primary service and also its own response are captured and interpreted by the Byzantine fault tolerant message receiver (BFTMessageReceiver). The BFTMessageReceiver which extends the AbstractInOutSyncMessageReceiver class of AXIS2 runtime engine is designed to analyze the messages that are transmitted by the participating services including the primary service. When all the responses of the participating services agree with each other, the BFTMessageReceiver forwards the response as generated by the primary service to the client without any modification. The service whose response does not agree with the responses of the other services is to be tested for its malicious behaviour. The responses of the primary service provider and other participating services are ensured by applying Lamport's [1] algorithm for Byzantine agreement. If it is found that the identified service is behaving abnormally, then the BFTMessageReceiver rectifies the fault and generates the response as expected by the client. Figure 1 depicts the proposed model for handling Byzantine faults in the SOAP communication framework.

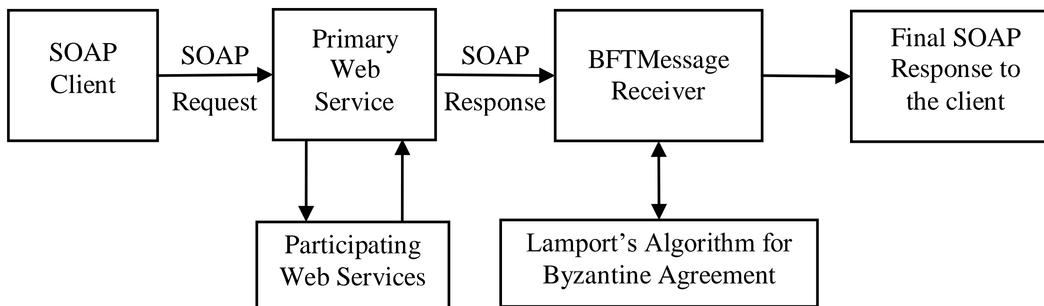

Fig. 1: Extended SOAP Communication Framework with Byzantine Fault Tolerance

The class "BFTMessageReceiver" is defined as a library file and included in the package "org.apache.axis2.receivers", which is associated with axis2-kernel. The class "BFTMessageReceiver" is specified as the preferred message exchange pattern in the service description file. The executeImpl() method of OperationClient class associated with AXIS2 client engine is capable of accessing the message receiver class from the "services.xml" file and launches the message context, which is sent by the client as request. This modified message exchange pattern is usually listened by the primary or controlling service. The BFTMessageReceiver class encapsulates the "invokeLamportBFT()" method which has to be invoked by the "invokeBusinessLogic()" that is inherent to any message receiver and the AXIS2 runtime is capable of calling it directly.

3.0 CASE STUDY – VIDEO on DEMAND SYSTEM

The Byzantine fault tolerance model in distributed services environment is tested with a Video on Demand system, which comprises of "VideoService" as a primary service and authentication, subscription and file search services as coordinating services. The Video on Demand system has been developed using the SOAP communication framework and deployed in the AXIS2 runtime environment. The client may send the request directly to the primary service and receive the response as per the Lamport's decision within the BFTMessageReceiver, which is an extension to the existing AXIS2 SOAP communication framework.

A set of interfaces has been defined for implementing the Lamport's algorithm for Byzantine agreement as a Web service named "invokeLamportBFT()" service. The set of interfaces used for implementing the Byzantine algorithm are as follows:

Service interface	:	maintains the list of participant services
NodeValue interface	:	retrieves current message value that is passed from one service to another service
MapRepository interface	:	defines the services hierarchy, identifies the path in which the message communication takes place, specifies the number of messages exchanged between the participant services
Broadcast interface	:	maintains the number of repetitions and the message transfer details

The "invokeLamportBFT()" service is invoked by the BFTMessageReceiver inherently only when one or more participating services exhibit traitorous behaviour. Otherwise, the BFTMessageReceiver executes similar to the original message receiver well defined within the SOAP communication framework.

In the proposed Video on Demand system, the Byzantine behaviour is injected into the "FileSearchService" using an aspect. The task of the aspect is to induce the fault generation code externally and makes "FileSearchService" to execute abnormally and this malicious aspect is hidden from the service. The fault generating aspect and the "FileSearchService" are weaved together through the pointcut "fileNotAvailable". Whenever the method "fileSearch()" of FileSearchService is invoked, the aspect pointcut "fileNotAvailable" is triggered, and this in turn, invokes the *around* advice. The objective of *around* advice is to take control of the method "fileSearch()" and makes it always dispatch a "false" response. The method signature of "fileSearch()" specifies a parameter of type String, hence, pointcut is also defined with a String parameter. The aspect code for the pointcut "fileNotAvailable()" with the associated *around* advice is given below:

```
public pointcut fileNotAvailable(String s): execution (public boolean VideoService.fileSearch(String)) && args(s);
boolean around(String s): fileNotAvailable(s)
{
 return false;
}
```

The method "fileSearch()" encapsulated within the FileSearchService to check the availability of the file requested by the client, which is controlled by the aspect's pointcut "fileNotAvailable()" It is defined as follows:

```
public boolean fileSearch(String filename)
{
	File f=new File(filename);
	if (f.exists())
		return true;
	else
		return false;
}
```

Since aspects do not require any external invocation because it executes along with the service, the "fileSearch()" method does not contain any explicit configuration information to invoke the aspect pointcut "fileNotAvailable".

The Video on Demand service with faulty "FileSearchService" that exhibits Byzantine behaviour is deployed in the Web application server. In the service description file for Video on Demand service, the message exchange pattern is assigned with the class "BFTMessageReceiver". The class BFTMessageReceiver extends the AXIS2 based SOAP communication message receiver AbstractInOutSyncMessageReceiver. It is defined within the package

org.apache.axis2.receivers and archived with the existing “axis2-kernel” core of the AXIS2 runtime. The configuration file (services.xml) to describe the Video on Demand service with the new message receiver is given below.

```
<service name="VideoAspectService">
<parameter name="enableSwA">true</parameter>
<messageReceivers>
<messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
      class="org.apache.axis2.receivers.BFTMessageReceiver"></messageReceiver>
</messageReceivers>
<parameter name="ServiceClass" locked="false">
         video.VideoService</parameter>
</service>
```

Figure 2 shows the proposed model for Byzantine fault tolerance in Video on Demand system. The video service is the primary service to which the client can send the SOAP request directly. The authentication service is the initial service invoked by the Video service to validate the user's credentials. The authentication process is followed by the Subscription service to determine the subscription status of the authorized user and decides whether access to the video content is allowed or denied. The FileSearchService checks for the existence of the requested file and always returns a “false” message since it is controlled by the aspect pointcut “fileNotAvailable”. Because the “FileSearchService” exhibits Byzantine behaviour, the VideoService does not immediately dispatch the requested video to the client.

A SOAP handler is written to maintain the sequence of services associated with the Video on Demand system in a predefined order. The SOAP request sent by the client is received by the service provider (VideoService) as a SOAP MessageContext. A handler chain is used to control the sequence of operations to obtain the desired results. Separate handlers are defined using <handler> elements for authentication service, subscription service and file search service within a handler chain.

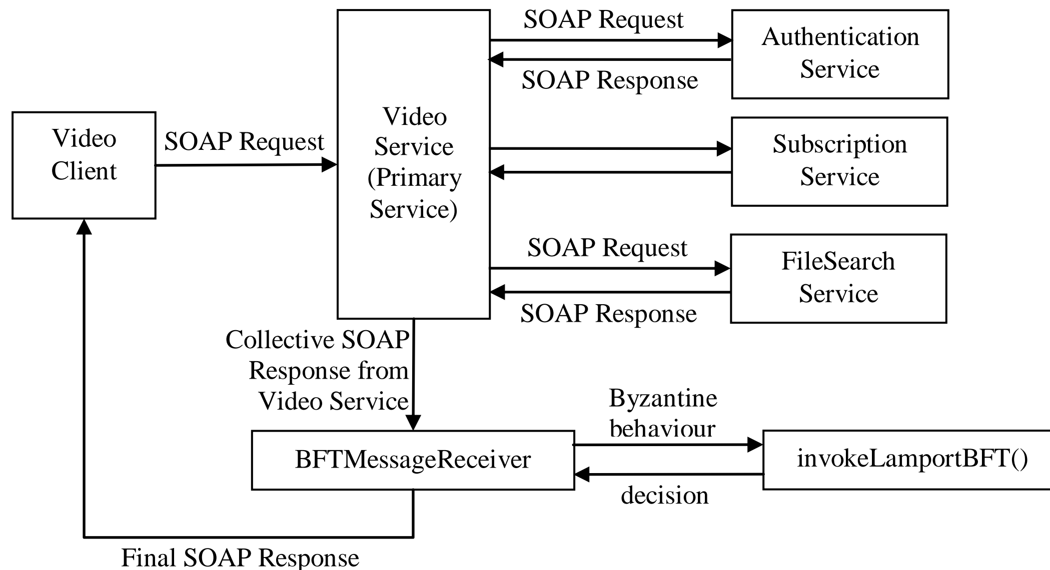

Fig. 2: Byzantine Fault Tolerant Video on Demand Model

The handler chain is associated with the Video on Demand service using the annotation @HandlerChain with the "file" element initialized with "handler-chain.xml". The Video on Demand service calls the handleMessage() method, which receives the request payload as SOAP MessageContext. All service handlers listed in sequence within the "handler-chain.xml" file are effectively managed by the handleMessage() method and each SOAP response is sent to the customized message receiver "BFTMessageReceiver".

The handler chain used in the Video on Demand model is defined in an XML file named as "handler-chain.xml" and it is given as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns=http://java.sun.com/xml/ns/javaee>
<handler-chain>
<handler>
<handler-name>Authentication Service</handler-name>
<handler-class>video.AuthenticationService</handler-class>
</handler>
<handler>
<handler-name>Subscription Service</handler-name>
<handler-class>video.SubscriptionService</handler-class>
</handler>
<handler>
<handler-name>File Search Service</handler-name>
<handler-class>video.FileSearchService</handler-class>
</handler>
</handler-chain>
</handler-chains>
```

The SOAP request sent by the client to the VideoService with the required video file and the authentication details is given as follows:

```
<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<video:getVideo xmlns:video="http://video">
<video:username>…</video:username>
<video:password>…</video:password>
<video:videoName>video.mpg</video:videoName>
</video:getVideo>
</soapenv:Body>
</soapenv:Envelope>
```

The authentication service sends a "true" message and the subscription service sends a "yes" message. As per the assumption, the file search service is the traitor and by considering its response, the video service returns a faulty message with the value "Video not available" to the BFTMessageReceiver. The response framed by the VideoService for the client's request is given as follows:

```
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<video:videoResp xmlns:video="http://video">
```

```
<video:authentication>true</video:authentication >
<video:subscription>yes</video:subscription>
<video:fileSearch>false</video:fileSearch>
<video:videoContent href="denied">Video not available</video:videoContent>
</video:videoResp>
</soapenv:Body>
```

If the responses of all coordinating services agree with each other, the response of the VideoService contains the requested video content as SOAP attachment along with its unique reference id in order to access the same and the SOAP response, thus formulated, is dispatched to the client. If any one of the services exhibits a traitorous behaviour, then the BFTMessageReceiver has to analyze the SOAP response with an exception message as sent by the VideoService and invokes the Lamport's Byzantine agreement procedure to ensure the genuineness of the client, and hence, to formulate the actual SOAP response as expected by the client. The new message receiver BFTMessageReceiver that effectively handles the Byzantine faults is defined as follows:

```
package org.apache.axis2.receivers;
public class BFTMessageReceiver extends AbstractInOutSyncMessageReceiver
implements MessageReceiver {
public void invokeBusinessLogic(MessageContext msgContext, MessageContext newmsgContext) throws AxisFault {
            // get the implementation class for the Web Service find the WebService method
               ………..
            //to invoke getVideo() method in VideoService
            OMElement result = (OMElement) method.invoke(obj, new Object[]
                                    {msgContext.getEnvelope().getBody().getFirstElement()});
            SOAPFactory fac = getSOAPFactory(msgContext);
            SOAPEnvelope envelope = fac.getDefaultEnvelope();
             if (result != null) {
                  envelope.getBody().addChild(result);
                  SOAPBody body1 = envelope.getBody();
                  OMElement element=body1.getFirstChildWithName(new QName("http://video",
                                                                              "videoResp"));
                  OMElement element1=element.getFirstChildWithName(new QName("http://video",
                                                                              "legitimate"));
                  String value1=element1.getText();
                  //get the response "value2" of Subscription Service
                  //get the response "value3" of File Search Service
                  OMElement element4=element.getFirstChildWithName(new QName("http://video",
                                                                              "videoContent"));
                  String value4=element4.getAttributeValue(new QName("href"));
                  // It is assumed that FileSearchService generates Byzantine behaviour
                  boolean decision=invokeLamportBFT(value1, value2, value3, value4);
                  if (decision) {
                  //The decision to commit the transaction is the outcome of Lamport's algorithm
                  //Rectify the faulty service and dispatch the modified SOAP response to the client
                        SOAPBody body2=msgContext.getEnvelope().getBody();
                        OMElement element5=body2.getFirstChildWithName(new QName("http://video",
                                                                              "getVideo"));
                        OMElement element6 = element5.getFirstChildWithName(new
                                                      QName("http://video", "videoName"));
```

```
                    String filename=element6.getText();
                    FileDataSource fds=new FileDataSource(filename);
                    DataHandler dh=new DataHandler(fds);
                    String videoID=newmsgContext.addAttachment(dh);
                    element4.removeAttribute(element4.getAttribute(new QName("href")));
                    element4.addAttribute("href", videoID, null);
                }
                else
                {
                    // No malicious behaviour detected. Agree with the response as generated by the
                    // primary service provider and forward the same to the client.
                }
                  newmsgContext.setEnvelope(envelope);
        }
}
```

The identified malicious service is the FileSearchService, and therefore, the BFTMessageReceiver takes the necessary steps to rectify the faulty response by executing the task of locating the file. It identifies that the requested video file exists and the modified response message dispatched by the BFTMessageReceiver is shown below.

```
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<video:videoResp xmlns:video="http://video">
<video:videoContent href="urn:uuid:D14831D69C6C569FFD1317129981605"/>
</video:videoResp>
</soapenv:Body>
```

The response messages of the participating services are not included in the SOAP response because it is not of concern to the client. The BFTMessageReceiver attaches the requested video file to the SOAP response and dispatches it to the client. In this model, the genuine client receives the intended response even in the presence of Byzantine faults.

4.0 CONCLUSION

In a real time system, where many Web services are coordinating to complete a specific task, it is hard to predict the services exactly those are behaving differently i.e., behaving as traitors. Hence 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 to encapsulate the Lamport's method for Byzantine Agreement without detrimental to the normal responsibilities of the original message receiver. The proposed model derives a concrete decision inherently using Lamport's algorithm for the action to be taken with the presence of traitors using the extended message receiver. The traitorous behaviour is injected into the model i.e., into one of the coordinating services using aspects. An enhanced model for identifying Byzantine behaviour of Web services using aspects is proposed in this work. The proposed approach is an inherent extension to the existing SOAP communication framework and this is very much essential that this mechanism is to be built within the core of the message exchange pattern in order to make the system be alert in the presence of vulnerable Byzantine faults. The model identifies the faulty service and discards the exception message by forming a fault free response to the client. The genuine client is assured with the delivery of content even in the presence of Byzantine faults. An additional overhead is involved for the new message receiver, as it has to invoke the Lamport's algorithm and it rectifies the fault in a genuine case as decided by the Lamport's algorithm. This overhead is not a major concern as the client

receives the intended response. It is also to be noted that the extended new message receiver should be free from Byzantine faults.

REFERENCES

[1] Leslie Lamport, Marshall Pease, Robert Shostak, "The Byzantine Generals Problem" *ACM Transactions on Programming Languages and Systems* Vol 4 No.3, July 1982, pp. 382-401.

[2] Wenbing Zhao, "BFT-WS: A Byzantine Fault Tolerance Framework for Web Services", in *11th International IEEE Conference Enterprise Distributed Object Computing Conference Workshops*, 2007, 15-16 October, Annapolis, Maryland, USA, pp. 89-96.

[3] Zhang Ji-de and Yao Ying, "Analysis of exception fault types based on AspectJ", in *International Conference on Computer Application and System Modeling,* 2010, 22-24 December, Taiyuan Shanxi, China, Volume I, pp. 287-289, IEEE.

[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", in *Sixth IEEE International Symposium on Network Computing and Applications*, 2007, 12-14 July, Cambridge MA, USA, pp. 195-204, IEEE.

[5] J. Herrero, F. Sanchez, and M. Toro, "Fault Tolerance AOP Approach", *International Workshop on Aspect-Oriented Programming and Separation of Concerns*, Lancaster University, UK, 24 August, 2001, pp. 44-52.

[6] AspectJ, http://eclipse.org/aspectj

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

[8] R. A. Bazzi and M. Herlihy, "Enhanced Fault-Tolerance through Byzantine Failure Detection", in *13th International Conference on Principles of Distributed Systems*, 2009, 15-18 December, Nimes, France, pp.129-143.

BIOGRAPHY

Murugan Sankaranarayanan received his M.E degree in Sathyabama University with University Rank. He is currently a Research Scholar in the Faculty of Computer Science and Engineering, Sathyabama University. Area of interest includes Web Technology, Cloud Computing and Service Oriented Architecture.

Ramachandran Veilumuthu received his M.E. degree and Ph.D. in Electrical Engineering from, Anna University, Chennai, India. He is currently working as a Professor in the Department of Information Science and Technology, College of Engineering, Guindy, Anna University, Chennai. His research interest includes Soft Computing, Web Technology, Service Oriented Architecture, Cloud Computing, Computer Applications in Power System Analysis and Network Security.