BYZANTINE FAULT TOLERANCE MODEL FOR SOAP FAULTS

S. Murugan

Research Scholar, Faculty of Computer Science and Engineering,
Sathyabama University, Chennai, Tamilnadu, India.
snmurugan@live.com

V. Ramachandran

Professor, Department of Information Science and Technology,
Anna University, Chennai, Tamilnadu, India.
rama@annauniv.edu

Abstract

The proposed model is to configure Byzantine Fault Tolerance mechanism for every SOAP fault message that is transmitted. The reliability and availability are of major requirements of Web services since they operate in the distributed environment. One of the reliability issues is handling faults. Fault occurs in all the phases of Service Oriented Architecture i.e. during publishing, discovery, composition, binding, and execution. These faults may lead to service downtime, behaves abnormally, and may send incorrect responses. These abnormalities are classified as Byzantine faults in Web services. Even though SOAP specification provides fault handling mechanisms, the correctness of the received SOAP fault messages are not known. In this paper, a model is proposed to check the correctness of the SOAP fault message received, by incorporating the Byzantine agreement for fault tolerance. The existing fault tolerant mechanism detects server failure and routes the request to the next available server without the knowledge of the client. The proposed model ensures a transparent environment by providing fault handling information to the client. This is achieved by incorporating an active replication technique.

***Keywords*:** Byzantine; Fault tolerance; Replication; SOAP; Web Services

1. Introduction

Web service is a heterogeneous architecture for deploying services in the Web. For effective communication, Simple Object Access Protocol (SOAP) is recognized as a more promising transport protocol for sending requests and receiving responses in Web services. A Web service is a piece of software application whose interface and binding can be defined, described, and discovered as XML artifacts.

Fault handling is a major issue that is been considered in many areas. Service-based applications should adopt fault handling mechanisms like fault tolerance to handle errors as propagated by the respective services, and ensuring an end-to-end Quality of Service (QoS). Failures may occur because of congestion in networks, server load, hardware fault at server end, software failure at server end, denial of service attack etc., Fault-tolerant frameworks performs fault detection and fault recovery. This is achieved by redundancy.

In case of Web services the cost for maintaining and recovery from faults is higher than any other normal applications and there may be faults that may happen within the Web services as well. Several fault tolerance approaches have been proposed for Web services, but the field still requires theoretical foundations, appropriate models, effective design paradigms, practical implementations, and in-depth experimentations for building highly-dependable Web services (Chan *et al*. (2006)). WS-Reliability and WS-ReliableMessaging specifications are dealt with faults at the transport level.As per the current research works reported, all the faults during the execution of the Web services are handled without the knowledge of the client and also, the client does not know whether the received response is an authenticated one. The proposed approach focuses on transparent fault handling mechanism by providing information to the client about the current status.

In general, when a fault is generated, no further processing should be performed. In request-response exchanges, a fault message will be transmitted to the sender of the request, and some application level error will be flagged to the user (WS-I Basic Profile 1.1). A fault message consists of fault code, fault message, fault detail, and fault actor.

Fault tolerance in Web services is achieved by replication and redundancy. In this paper we are applying replication technique for achieving fault tolerance. Two common replication strategies are: (1) passive replication and (2) active replication. Passive replication employs a primary replica to process the service request first and invokes backup replicas only when the primary replica fails. Active replication invokes all replicas at the same time and employs the first properly returned response as the final outcome (Salas (2006)).

The incorporation of Byzantine Fault Tolerance (BFT) in Web services will address the issues of a) increase in malicious attacks in Web applications, b) software faults, c) the growing values of data, and d) provides a non fail stop behaviour of services ((Castro and Liskov (1999)), Kotla *et al.* (2008)). Reduction in the costs of hardware and effective practical implementation of BFT replication makes the service providers to incorporate into their web servers. For example by default, the Google file system uses 3-way replication of storage, which is roughly the cost of BFT replication for f = 1 failures with 4 agreement nodes and 3 execution nodes (Yin *et al.* (2003)).

2. Fault Tolerance Mechanisms

Different phases involved while implementing Web services using Service Oriented Architecture are publishing, discovery, composition, binding, and execution. Faults may occur during any stage and their categories and fault types are given in Table 1.

Table 1. Fault Categories in Web Services

Fault Classification	Fault Types
Publishing	Service Description, Service Deployment
Discovery	No service found, Wrong service, timed out
Composition	Incompatible components, timed out
Binding	Authentication failure, Accounting problems, timed out
Execution	Server crash, Incorrect result, timed out

Aghdaie and Tamir (2001) developed a client transparent fault tolerance model for Web servers. It detects server errors and routes the requests to a standby backup server for reducing service failures. G. T. Santos et.al (2005) describes active replication technique in Web services. To support consumer transparent fault tolerance, the architecture has a central forwarding component that includes the mechanisms responsible for managing replicas. It acts as a broker between consumers and providers. To deal with the fault of the forwarding component, the architecture has a backup component. In this architecture, the broker is implemented as a Web service and it enables consumers to use different brokers. The forwarding component has a configuration system for creating groups and uses UDDI tModels for creating service replica groups. The passive replication technique is explored in (Liang *et al.* (2003)). To achieve fault tolerance, some modifications are proposed for the WSDL and SOAP standards with the purpose of allowing the specification of Web service replicas and redirection of service requests. This approach employs a mediator layer in the Web service architecture, which offers the necessary functionalities for managing replicas. Thus, the passive replication approach does not depend on modifications in the interface description language and the message interchange protocol of Web services. The passive replication approach includes a UDDI extension, but the extended UDDI is compatible with the standard. WS-Replication (Salas (2006)) is a framework for seamless active replication of Web services, that is, it respects Web service autonomy and provides transparent replication and failover. The communication within the framework is exclusively based on SOAP without forcing the use of an alternative communication means. WS-Replication allows the deployment of a replicated Web service using SOAP for transporting information across sites. Clients invoke a replicated Web service in the same way as they invoke a non-replicated one. Internally, WS-Replication takes care of deploying the Web service in a set of sites, transparently transforming a Web service invocation into a multicast message to replicate the invocation at all sites, awaiting for one, a majority or all replies, and delivering a single reply to the client (Salas (2006)).

Faults may be classified as transient faults, intermittent faults, and permanent faults. The system faults can be fail-silent failures and byzantine failures. These faults can be overcome by applying redundancy and replication of services. In this paper, the SOAP faults at the execution phase and Byzantine failures are being dealt with. The replication approach is used for fault tolerance. The faults are received as SOAP fault messages as received as a payload.

3. SOAP Faults

In Web services, when a fault occurs during the service execution an appropriate SOAP fault message is sent to the requester. Fault is an optional sub element of a SOAP Body. This element is been included to intimate the

receiver with error messages and can also be parsed like any other XML elements. Each Fault element is characterized using:

faultCode	: Unique text code for the error.
faultString	: To specify the error in detailed manner
faultActor	: A string element to specify the root cause of the fault.
detail	: Specifies the application specific error messages.

The general format of a SOAP Fault is given as below.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <env:Body>
        <env:Fault>
            <faultcode xsi:type="xsd:string">env:Client</faultcode>
            <faultstring xsi:type="xsd:string">The root cause of the error</faultstring>
            <faultactor>http://www.abccorp.com</faultactor>
            <detail>Detailed error description</detail>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

The fault code values as per the SOAP specification 1.2 are VersionMismatch, MustUnderstand, DataEncodingUnkown, Sender and Receiver. The fault codes Client and Server in SOAP 1.1 specification is renamed as Sender and Receiver in SOAP 1.2 specification. Each category of generic SOAP faults is analyzed for tolerating. The above said fault codes are sent to the client node and to be analyzed, by applying Byzantine agreement algorithm.

4. Byzantine Generals Problem

In 1982, Leslie Lamport described Byzantine general's problem in a paper written with Marshall Pease and Robert Shostak (1982). This problem is built around an imaginary General who makes a decision to attack or retreat, and must communicate the decision to his lieutenants. A given number of these actors are traitors (possibly including the General.) Traitors cannot be relied upon to properly communicate 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, and the orders sent to the other processes are messages. Traitorous generals, and lieutenants are faulty processes, and loyal generals, and lieutenants are correct processes. The order to retreat or attack is a message with a single bit of information: a one or a zero.

Byzantine fault tolerance (Lamport *et al.* (1982)) is a highly reliable replication technique that can tolerate a wide range of types of faults. Under a Byzantine fault model, a faulty node may act arbitrarily; this is in contrast to the more restrictive crash-fault model, which assumes that a faulty node will simply be unresponsive. Although only f+1 replicas are required to tolerate f Byzantine faults if communication is assumed to be synchronous and replicas can authenticate messages, this synchrony assumption is not practical for environments like the Internet, where there is no a priori known upper-bound on communication latencies (Merideth *et al.* (2005)). Castro–Liskov (1999) proposed a Byzantine Fault Tolerance state-machine-replication protocol (Liang *et al.* (2003)) and its library implementation can be used to create client–server Byzantine-fault-tolerant applications. This technique works in asynchronous environments. BFT-WS (Zhao (2007)) architecture is implemented as an Axis2 module. During the out-flow of a SOAP message, Axis2 invokes the BFT-WS Out Handler during the user phase, and invokes the Rampart (Nadalin *et al.* (2004))handler for message signing during the security phase. Then, the message is passed to the HTTP transport sender to forward to the target endpoint. During the inflow of a SOAP message, Axis2 first invokes the default handler for preliminary processing (to find the target object for the message based on the URI and SOAP action specified in the message) during the transport phase, it then invokes the Rampart handler for signature verification during the security phase. This is followed by the invocation of the BFT-WS Global In Handler during the dispatch phase. This handler performs tasks that should be done prior to dispatching, such as duplicate suppression at the server side. If the message is targeted toward a BFT-WS-enabled service, the BFT-WS In Handler is invoked for further processing during the user-defined phase, otherwise, the message is directly dispatched to the Axis2 message receiver. Michael et.al (2005) presents a Thema which consists of a client library (Thema-C2RS), a BFT service library (Thema-RS), and an external service library (Thema-US). Each of these libraries provides support for the standard multi-tier Web Service programming model and SOAP communication.

5. Handling SOAP Faults with Byzantine Agreement

The proposed work analyzes the faults that are received from various replicas by applying Byzantine agreement. Whenever there is a fault, the client receives the SOAP Fault from all the replicas, applies Byzantine agreement, identifies the erroneous replica and eliminates it. The processes involved in this model are replication management and eliminating the faulty replica.

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 of the services 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. 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.

A replica manager is developed for managing and controlling the Web services. When the primary server is crashed the client's request is forwarded to other available replicas. The participating replica may sometimes behave abnormally by generating illegitimate response. The response generated by this faulty replica is dispatched to the client. Faulty responses may lead to business loss, customer dissatisfaction, loss of reputation and various other factors. To avoid the problems of this nature, message from all the replicas are received for identifying the presence of Byzantine fault and to eliminate the faulty service.

An enhanced methodology is adopted for handling Byzantine faults in the SOAP fault messages. Figure 1 shows replication manager that coordinates primary and secondary services. Replication (Liu 2008) is the redundant fault tolerant mechanism adopted in distributed systems to make the service available to the client even in the presence of crash. In the proposed model, the replica manager requests controls the replicas of the server. The replicas that offer similar kind of services are grouped together and referred to as "Service Group". The service that receives requests from the client is referred to as "primary service". When the primary service is crashed the control is transferred to the replication manager.

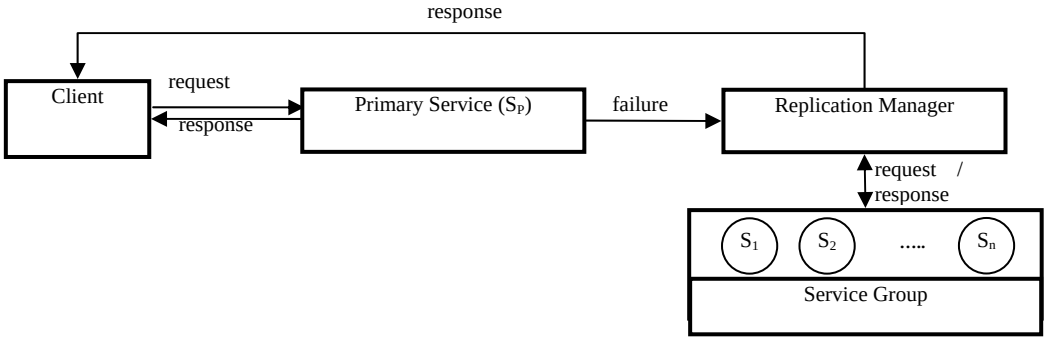

Fig. 1. Replication Management

S_i – Secondary Replicas

The replication can be passive or active. The replication manager uses active replication technique for processing the requests when the primary server is crashed. In active replication technique, all the replicas receive the request, process the request and forward the response to the replication manager. As each replica service sends a response, the replication manager has to make a decision (referred to as agreement) on the values received. In the received responses there may be some replica(s) exhibit Byzantine behaviour by generating faulty responses. The replica manager is to identify the faulty replica and eliminate the same by applying Lamport's algorithm for tolerating Byzantine faults. Once these faults are eliminated the client receives the fault free response.

When all the services agree upon an exception message, SOAP fault is the response sent to the client. In some situations the service that exhibit Byzantine behaviour dispatches an exception message that is similar to non-faulty services. In this situation the faulty and non-faulty services cannot be distinguished as all the services throw an exception message and it becomes difficult to identify the abnormal service. To eliminate the faulty node, the SOAP fault messages are to be analyzed. The Web services may be developed using any of the technology, Java based application programming interfaces like Remote Method Invocation (RMI), Enterprise Java Beans (EJB), Web Servlets, Java Server Pages (JSP), Model View Controller (MVC) using Struts, Hibernate etc., Microsoft based languages like Visual Basic, Visual C++, ActiveX Server Pages (ASP), .NET Framework etc., or any other programming languages like PERL, COBOL etc., All these programming

paradigms are capable of throwing exceptions and these exceptions are represented as SOAP faults. The proposed methodology is adaptable for Web services developed using any programming approach.

6. Classification of SOAP Faults

Each category of generic SOAP Fault has been analyzed and the application of Byzantine agreement algorithm is detailed as follows.

6.1. MustUnderstand

When a SOAP message is received by the node (intermediate or ultimate receiver) it has to examine Header element to check any intermediate processing is to be done here in this node. This is applicable if the mustUnderstand attribute is set to "1". If the node is not recognizing the header block then the fault code MustUnderstand is generated. The SOAP message of MustUnderstand is given below:

```
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
    <env:Body>
        <env:Fault>
            <faultcode>env:MustUnderstand</faultcode>
            <faultstring>Mandatory header block not understood.</faultstring>
            <faultactor>Faulty URI</faultactor>
            <detail>header</detail>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

Many intermediate services are involved in processing a request. Depends on the requirement some of the intermediary services may be ignored and in some situations the request should be processed by these intermediary services. When the intermediary services unable to process the request the MustUnderstand fault is triggered. In any system before processing the client's request the authentication details of the client is verified. The following code fragment shows the header of the SOAP request message by incorporating the MustUnderstand attribute.

```
<env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
    <env:Header>
        <vendor:authentication xmlns:vendor="urn:creditcard:vendor" soap:mustUnderstand="1">
        <vendor:vendorname>Name of the Vendor</vendor:vendorame>
        <vendor:vendorcode>Vendor ID</vendor:vendorcode>
        </vendor:vendor>
    </env:Header>
</env:Envelope>
```

When the authentication service is unable to validate the client's identity the following message is generated by the intermediate processing service.

```
<faultcode>env:MustUnderstand</faultcode>
<faultstring>Mandatory header block not understood.</faultstring>
<faultactor>urn:creditcard:vendor</faultactor>
<detail>UnknownVendor</detail>
```

6.2. VersionMismatch

The VersionMismatch SOAP fault code is generated whenever there is a difference in the namespace of envelope element of SOAP message. In SOAP envelope the namespace "http://www.w3.org/2001/12/soap-envelope" specifies the SOAP version 1.2. The elements supported in SOAP version 1.2 are either not available or not compatible with SOAP version 1.1. When there is a SOAP version mismatch between the client and request the following message is dispatched.

```
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
    <env:Body>
        <env:Fault>
            <faultcode>env:VersionMismatch</faultcode>
```

```
            <faultstring>1.1</faultstring>
            <faultactor>URI of Faulty Node</faultactor>
            <detail>Message was not SOAP 1.1-conformant</detail>
        </env:Fault>
    </env:Body>
<env:Envelope>
```

In the intermediary services, if the message transferred by the node adopts SOAP version 1.2 and the receiving service works with SOAP 1.1 version then the fault VersionMismatch is dispatched.

6.3. DataEncodingUnknown

The SOAP fault code DataEncodingUnknown is available in SOAP Version 1.2. In the SOAP request or response message when the XML is unable to parse the message content then the SOAP fault DataEncodingUnkonwn is triggered.

```
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
    <env:Body>
        <env:Fault>
            <faultcode>env:DataEncodingUnknown</faultcode>
            <faultstring>Nonconformance to WSDL definition</faultstring>
            <faultactor>URI of the faulty node</faultactor>
            <detail>Data Encoding Format not recognized</detail>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

Several encoding formats are available for handling SOAP messages. If the intermediary node is not capable of parsing these encoding formats then the SOAP fault DataEncodingUnknown is generated.

6.4. Sender

When incorrect SOAP request message is dispatched by the client then the fault code “Sender” of SOAP version 1.2 is dispatched. In SOAP version 1.1 this fault is identified as “client”. The fault code “Sender” is usually generated when the message transferred with lack of information or invalid data. The fault code specifies the root cause of the error message.

```
<env:Fault>
<faultcode>env:Sender</faultcode>
<faultstring>Invalid data</faultstring>
<optional sub codes can be created for specifying the detailed fault information/>
</env:Fault>
```

In a Credit card validating service, the SOAP request sent by the client should carry valid credit number as per the required format. The request is not validated by the service provider when the request message contains any special characters and the response carries the fault code “Sender”.

```
<env:fault>
<faultcode>env:Sender</faultcode>
<faultsubcode>exh:invalidCharacters</faultsubcode>
<faultstring>Card number contains invalid characters</faultstring>
<faultdetail>Contains Special Characters </faultdetail>
</env:fault>
```

The response message also carries the specific reason pertaining to the fault.

6.5. Receiver

The requested message could not be processed for several reasons. One such reason is that the intermediary node may not available for processing the request at that instance. When the intermediary service is available the response may be processed even at a later stage.

```
<env:Fault>
```

<faultcode>env:Receiver</faultcode>
<faultstring>Unable to process</faultstring>
<optional sub codes can be created for specifying the detailed fault information/>
</env:Fault>

When the intermediary node for verifying the balance of a customer's account in a bank is not available then the response to the client is as follows:

<faultcode>env:Receiver</faultcode>
<faultstring>Unable to find the service balanceCheck</faultstring>
<faultreason>Online Service is down for maintenance. Try later</faultreason>

The response message carries the fault message with the appropriate reason to the client. When the replication manager receives the responses from the replicas, the replication manager has to analyze the received response. The received responses are received into four categories. As a first case, when all the replicas have sent the responses and if all the values are equal then no further processing is required i.e. all the replicas are producing similar results and the services are not exhibiting Byzantine behaviour. In the second case when all the received values are not equal, then many replicas are producing incorrect results and by applying Byzantine agreement algorithm faulty replicas are identified and eliminated. In the next case, when a response from one of the replica is not matching with the others (where other replicas produces similar results) then the replica that produces a different response is the identified as fault replica. Finally, when some of the replicas are producing different values and some replicas producing equal results then apply Byzantine agreement algorithm and identify the faulty replicas. The procedure is given as follows:

(1) Collect the entire SOAP Fault from the replicas and perform the following:

case i: ((all) $SF_i = SF_j$)
all the replicas produced similar results. No further processing is required.
case ii: ((all) $SF_i <> SF_j$)
Byzantine failure occurred; all the replicas are producing incorrect results. Apply Lamport's algorithm Byzantine agreement and identify the faulty replicas.
case iii: ((any of) $SF_i <> SF_j$)
eliminate Rj being the replica producing incorrect result.
case iv: ((some of) $SF_i <> SF_j$)
Apply Byzantine agreement algorithm and eliminate the fault replicas.

(2) select maximum((equal)SF_i) as the correct response and dispatch it to the client.

where
R_i – i^{th} replica
SF_i is soap fault value that is received from the replica R_i.

The replication manager receives different set of values for every type of SOAP fault. The value of SFi depends on the type of SOAP fault message. In MustUnderstand SOAP fault the intermediate node (actor) is responsible for generating the fault and hence the element "fault actor" is returned by the replica. For VersionMismatch SOAP fault, "fault string" is returned as it carries the SOAP version (1.1 or 1.2). The DataEncodingUnknown SOAP fault, it is the intermediate node which is unable to process and hence the "fault actor" is returned. For sender and receiver SOAP fault messages detailed fault message is required. Hence for these SOAP faults the fault sub codes are returned as response messages. The procedure for handling the messages of various SOAP faults for each replica is given as follows:

if soapfault = env:MustUnderstand then
SF_i = (faultactor) SOAPFault.mustUnderstand(R_i)
else if soapfault = env:VersionMismatch then
SF_i = (faultstring) SOAPFault.versionMismatch(R_i)
else if soapfault = env:DataEncodingUnknown then
SF_i = (faultactor) SOAPFault.dataEncodingUnknown (R_i)
else if soapfault = env:Sender then
SF_i = (faultdetail) SOAPFault.sender(R_i)
else if soapfault = env:Receiver then
SF_i = (faultdetail) SOAPFault.receiver(R_i)

The response values of SF_i are transmitted to the replication manager for identifying and elimination of Byzantine behaviour among the replicas. Figure 2 depicts the flow of faulty messages from the replicas to the replication manager to determine the presence of Byzantine faults.

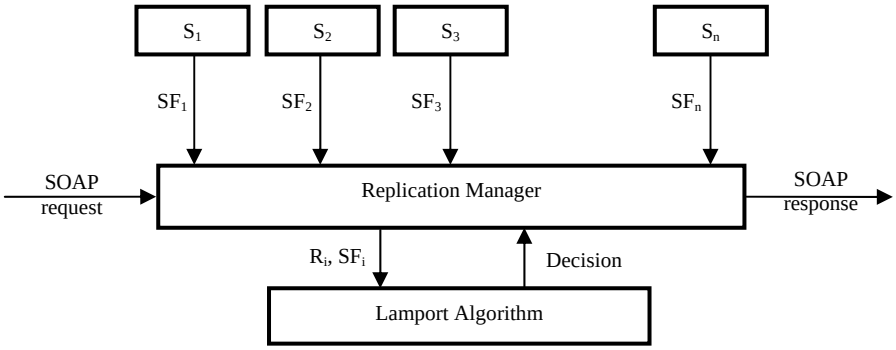

Fig. 2. Byzantine Agreement for SOAP Faults

R_i – i^{th} Replica
SF_i – SOAP Fault received from the i^{th} replica.
BFT – Lamport's Algorithm for Byzantine Fault Tolerance

The replication manager is capable of reaching the agreement when there is response from many replicas and they are not capable of handling Byzantine faults. The proposed approach reaches an agreement and transmits the fault free response to the client even in the presence of Byzantine faults.

7. Implementation

A set of interfaces has been created for implementing the proposed approach for handling Byzantine faults by applying Lamport's algorithm. The interfaces are ServiceFactoryInterface, FaultNotifierInterface, SOAPFaultInterface, ReplicaManagerInterface and BFTInterface.

7.1 ServiceFactoryInterface

The interface defines a ServiceFactory, to manage the list of services that offer similar kind of applications. The interface is capable of creating a new service, activates or deactivates a service and undeploys the service. For each service a unique ID is allotted. The applications are implemented using SOAP communication model and deployed in a Web Application server.

```
interface ServiceFactoryInterface {
  Service serviceCreate(serviceID);
  Status serviceActivate(serviceID);
  Status serviceRemove(serviceID);
};
```

7.2 ReplicaManagerInterface

ReplicaManager interface manipulates the replicas by adding, removing and the location of the replica server is maintained. Each and every replica is assigned with unique identification number. To create replicas and manage them, a HTTP Web server is used. The system is configured tested with both vertical and horizontal replication techniques. For vertical replication the replicaLocation method in the interface is not required.

```
interface ReplicaManagerInterface {
  Replica addReplica(replicaID);
  Replica  removeReplica(replicaID);
  Location replicaLocation(replicaID);
};
```

7.3 FaultNotifierInterface

The interface FaultNotifier identifies the malfunctioning replica and faulty replica. The replica ID of the faulty replica is returned.

```
interface FaultNotifier {
  Replica faultyReplica(replicaid);
};
```

7.4 BFTInterface

The interface BFT applies Lamport's algorithm for Byzantine fault tolerance. Replicas are created for all the nodes and messages are transmitted among themselves. All the broadcast messages are processed and the faulty process (replica id) is identified and reported to the replication manager.

```
interface BFT {
  Node addNode(replicaID);
  Message broadcastMessages (Message, replicaID);
  process(Message);
  Replica faultyNode(replicaID);
};
```

7.5 SOAPFaultInterface

The SOAPFault interface retrieves the faults that are generated by the replicas. The methods defined are for returning the faultactor element for mustUnderstand and dataEncodingUnknown, faultdetail for versionMismatch, sender and receiver SOAP faults.

```
interface SOAPFaultInterface {
  faultactor  mustUnderstand();
  faultactor dataEncodingUnknown();
  faultstring versionMismatch();
  faultdetail sender();
  faultdetail receiver();
};
```

8. Conclusion

The algorithm for handling the SOAP Faults in transparent manner by applying Byzantine algorithm is been proposed. For every type of SOAP Fault generated is analyzed and is processed accordingly. By using this approach the faulty replicas are eliminated and the client is assured with the authenticity of the received message. The client side overhead in processing the BFT algorithm is shown by varying the number of replicas. The number of messages that are communicated between the replicas is also shown. This model works in a blocked manner. This model may be extended to work in an asynchronous manner. In this model, the messages transmitted between the replicas are not signed one, the same algorithm may be extended to work for signed messages. As this model involves collection of results from all the replicas which leads to redundancy, complexity of the algorithm and increased computation cost. Hence the solution proposed is only for mission critical applications in which the cost of doing this process is well justified.

References

[1] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. (2004): Web Services Security: SOAP Message Security 1.0. OASIS, oasis standard 2004, 01 edition.

[2] D. Liang, C.-L. Fang, C. Chen, F. Lin. (2003): Fault tolerant Web service, in the proceedings of the 10th Asia-Pacific Software Engineering Conf., pp. 310–319. IEEE-Computer Society.

[3] G. T. Santos, L. C. Lung, C. Montez. (2005): FT-Web: A fault tolerant infrastructure for Web services in the Proceedings of the 9th IEEE International EDOC Enterprise Computing Conference, pp. 95–105. IEEE Computer Society.

[4] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. (2003): Separating agreement from execution for byzantine fault tolerant services, in proceedings of 19th ACM symposium on Operating Systems Principles (SOSP), Vol. 37, Issue 5.

[5] Jorge Salas, Francisco Perez-Sorrosal, Marta Patino-Martınez, and Ricardo Jimenez-Peris. (2006): "WS-Replication: A Framework for Highly Available Web Services, in 15th International Conference on the World Wide Web (WWW'06), pp. 357-366.

[6] L. Lamport, R. Shostak, M. Pease. (1982): The Byzantine generals problem, ACM Transactions on Programming Languages and Systems, 4(3) pp. 382–401.

[7] Michael G. Merideth, Arun Iyengar, Thomas Mikalsen, Stefan Tai, Isabelle Rouvellou, Priya Narasimhan. (2005): Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications, in the proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), pp:131-142.

[8] Miguel Castro and Barbara Liskov. (1999): “Practical Byzantine Fault Tolerance, in the Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, New Orleans, USA.

[9] Navid Aghdaie and Yuval Tamir. (2001): Client-Transparent Fault-Tolerant Web Service, in the proceedings of 20th IEEE International Conference on Performance, Computing, and Communications, (IPCCC’01), pp: 209-216.

[10] Pat. P.W. Chan, Michael R. Lyu, Miroslaw Malek. (2006): Making Services Fault Tolerant, the 3rd International Service Availability Symposium (ISAS 2006), Helsinki, Finland, Lecture Notes in Computer Science, vol. 4328, Springer, pp. 43-61.

[11] Qi Yu, Xumin Liu, Athman Bouguettaya, Brahim Medjahed. (2008): Deploying and managing Web services: issues, solutions, and directions, The VLDB Journal Volume 17, Issue 3: pp: 537 – 572, Springer-Verlag.

[12] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. (2008): Zyzzyva: Speculative Byzantine Fault Tolerance, in the proceedings of the Communications of the ACM Vol 51, Issue 11, pp: 86-95.

[13] SOAP Version 1.2, http://www.w3.org/TR/soap12-part1/.

[14] Stefan Bruning, Stephan Weissleder, Miroslaw Malek. (2007): A Fault Taxonomy for Service-Oriented Architecture in the proceedings of 10th IEEE High Assurance Systems Engineering Symposium (HASE'07), pp:367-368.

[15] Wenbing Zhao. (2007): BFT-WS: A Byzantine Fault Tolerance Framework for Web Services, in the proceedings of 11th International IEEE Conference Enterprise Distributed Object Computing Conference Workshops (EDOCW'07), pp: 89 – 96.

[16] WS-I Basic Profile 1.1 http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html