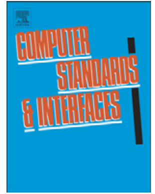
Computer Standards & Interfaces 36 (2013) 209–218

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi



A portable interceptor mechanism for SOAP frameworks

Chien-Cheng Lin [a,d], Chen-Liang Fang [b], Deron Liang [c,d,*]

[a] Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan, R.O.C.
[b] Department of Information Management, Jinwen University of Science and Technology, Taipei, Taiwan, R.O.C.
[c] Department of Computer Science and Information Engineering, National Central University, Taoyuan, Taiwan, R.O.C.
[d] Software Research Center, National Central University, Taoyuan, Taiwan, R.O.C.

ARTICLE INFO

Article history:
Received 31 May 2010
Received in revised form 11 December 2012
Accepted 4 April 2013
Available online 25 April 2013

Keywords:
SOAP framework
Web Service
Portable interceptor mechanism
ISO/IEC 9126

ABSTRACT

An interceptor is a generic architecture pattern, and has been used to resolve specific issues in a number of application domains. Many standard platforms such as CORBA also provide interception interfaces so that an interceptor developed for a specific application can become portable across systems running on the same platform. SOAP frameworks are commonly used platforms to build Web Services. However, there is no standard way to build interceptors portable across current SOAP frameworks, although, some of them provide proprietary interceptor solution within individual framework, such as Axis, XFire, and etc. In this paper, we propose the portable interceptor mechanism (PIM) consisting of a set of application programming interfaces (API) on SOAP engine, a core component of a SOAP framework. An interceptor is able to receive messages passing through the SOAP framework from the SOAP engine via these APIs. Furthermore, the proposed PIM facilitates run-time lifecycle management of interceptors that is a crucial feature to many application domains but is not fully supported by CORBA standard. For concept proving, we implement the proposed PIM on two popular SOAP frameworks, namely, Axis and XFire. We also discuss a number of implementation issues including the performance and reliability of PIM.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

An interceptor is a generic architecture pattern, and has been used to resolve specific issues in a number of application domains, such as continuous audit (CA), computer forensics (CF), fault tolerance (FT), load balancing (LB), and quality of service (QoS) systems. In the Continuous Process Auditing System (CPAS), an auditing module has been added to the original framework to intercept and log transactions so that the system's audit trail can function in a continuous manner [19]. Rezaee et al. suggested embedding an auditing module in a transaction system to monitor the system's activities continuously so that fraud could be detected in real-time [28]. Jailani et al. implement agent-based mechanism (i.e. AENA) to monitor and to record the activities related to transactions for auditing purpose [17]. For computer forensics applications, Sommer proposed using an intrusion detection system (IDS) to intercept and log system activities as digital evidence sources [32]. Song's dSniff program also employs the interception approach in computer forensics auditing [31]. Patel further mentions that an interception mechanism could be used to track users and their behavior for forensics purpose [27]. For fault-tolerance software applications, DOORS applies the interception approach to achieve fault transparency [22]; and Enternal uses interceptors for group communications in an active replication FT system [21]. Liang's work on fault tolerant Web Services shows that an interceptor mechanism is needed to solve logging, client fault transparency, and redundant nested invocation problems in fault tolerant systems [9,20]. Arpaia et al. develop a fault detection subsystem, which also use interception mechanism to extract relevant fault information [2]. For load balancing applications, Cao et al. and Hallstrom et al. implemented an interceptor in their agent-based resource management system to intercept user requests and distribute the load fairly over all grid servers [6,13]; while Othman et al. used a client-side interceptor to intercept and then forward requests for load balancing purposes [26]. In the QoS domain, Artaiam and Senivongse use an interception mechanism as a wrapper to monitor the system resources [3]; while Zeng et al. use the Self-Serv system as a wrapper to intercept messages [43]. Helali et al. use a middleware as a wrapper to perform QoS monitoring and managing [14].

Web Service is an increasingly recognized technology for building distributed applications where Simple Object Access Protocol (SOAP) [36,38] is one of the most commonly used means of implementation. Since interceptors have been widely used in various applications, we believe that Web Services would have greater applicability in various application domains if there were an interceptor mechanism for SOAP frameworks. Currently, two approaches are available for obtaining the messages flowing in and out of SOAP frameworks: the intermediary mechanism supported by the SOAP 1.2 [38] standard and proprietary interception supported by individual SOAP framework vendors [1,7,18]. The intermediary mechanism is operated as a midpoint proxy (or an intermediary node) and can be used to re-route and to pre-process messages between client and server nodes. Many applications

* Corresponding author. Tel.: +886 3 4227151x35208; fax: +886 3 4227151.
E-mail address: drliang@csie.ncu.edu.tw (D. Liang).

0920-5489/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.csi.2013.04.001

such as CA, CF, and FT require information interception at endpoints – either client-side or server-side – and some of them even require message interception at both endpoints, such as QoS. The intermediary mechanism, however, does not quite meet the requirements of these applications (further discussion can be found in Section 5.1). In contrast to the protocol layer approach as the intermediary mechanism, some SOAP frameworks, such as Apache Axis [1] and Codehaus XFire [7], provide proprietary application programming interfaces (APIs) on their SOAP engines with which an application developer is able to plug in application specific interceptors to obtain messages from the SOAP engines as they pass through. One of the drawbacks to taking this approach is the portability, i.e., an interceptor developed for Axis requires some effort to be ported to XFire, and vice versa. One possible way to resolve these issues is through an application platform (OS/middleware) that provides an interceptor mechanism with standard interfaces, such as the Common Object Request Broker Architecture's (CORBA's) portable interceptor (PI) [25] at the middleware layer and dSniff [31] at the network layer. However, CORBA PI falls short of lifecycle management at run-time (discussed in Section 5.2), which is a crucial functionality for applications such as continuous assurance and load balancing. These applications often need this managerial function to dynamically reconfigure their inner states without perturbing the operations on the original systems. We, therefore, propose a set of standard interfaces, called the portable interceptor mechanism (PIM), for SOAP engines that address the above issues. Notice that this work is based on our previous work [10].[1]

The research methodology of this paper is threefold. First, we determine the requirements of a portable interceptor mechanism (PIM) based on the specifications of several major application domains, such as the CA, CF, FT, LB, and QoS systems, and categorize them by using ISO/IEC 9126 [15]. Second, based on the requirements, we design two PIM interfaces: an interceptor management mechanism (IM) interface and a portable interceptor (PI) interface. The IM interface, which CORBA does not provide, allows interceptor administrators to manage the lifecycle of interceptors at run-time and also facilitates the interception of messages. The proposed PI interface can be used by developers to implement their own interceptors. Finally, we analyze the feasibility of applying the proposed PIM via a proof of concept. Specifically, we implement two prototypes and evaluate them based on the determined requirements.

The remainder of this paper is organized as follows. In Section 2, we define four PIM requirements based on the specifications of several major application domains and categorize them by ISO/IEC 9126 [15]. In Section 3, we analyze the PIM's features via a number of use cases and discuss the proposed PIM interfaces in detail. We investigate the feasibility of applying the PIM via a proof of concept in Section 4; and discuss two alternative approaches, namely, SOAP intermediary [38] and CORBA PI [25] in Section 5. We then summarize our findings in Section 6.

2. Determining PIM requirements

To design an effective portable interceptor mechanism (PIM), we must first determine the PIM requirements. We therefore study how interceptors are used in various application domains, such as continuous auditing (CA), computer forensics (CF), fault tolerance (FT), load balancing (LB), and Quality of Service (QoS) systems. To ensure that the requirements are addressed completely, we use ISO/IEC 9126 [15] to categorize them. ISO/IEC 9126, which is the evaluation standard recommended by the International Organization for Standardization, is used to assess the quality of an information system based on six quality factors, namely, functionality, usability, reliability, portability, maintainability, and efficiency. In the following, we define four PIM requirements and categorize them based on the above quality factors.

2.1. Functionality and portability

The interceptor approach, which is used to intercept messages exchanged between systems, is employed in many application domains. For example, Woodroof et al. suggested that a CA monitoring mechanism should be run in real time (or near real time) in order to track the necessary audit trail [41], while Sommer considered that a continuous logging system at both client and server sites is necessary for live forensic purposes [32]. Gunestas et al. used handlers to implement the Forensic Web Services (FWSs) framework for computer forensic domain [12]. Liang et al. proposed a fault-tolerant logging mechanism to continuously record all service invocations and messages for use in recovery operations [20]; while Felber and Narasimhan noted that the interceptor approach is one way to implement fault-tolerant systems [11]. The interception approach is also used to monitor system resources and intercept messages in the QoS domain [3,43]. Furthermore, most of these applications require information/message interception simultaneously at both client and server sites. Therefore, based on the above observations we define the first system requirement for the proposed PIM:

R1 A PIM should facilitate interaction between the interceptor and the SOAP engine so that designated interceptors can process SOAP messages, request messages and response messages that pass through the SOAP engines at the client and server sites continuously.

Based on our review of the literature, we believe that, although many application domains adopt the interceptor approach, it would be more useful if each interceptor could adapt to different domains and/or frameworks. For example, an interceptor implemented to resolve a fault tolerance issue could also be adapted to another framework applied by the CA mechanism. Therefore, we define the second system requirement:

R2 A PIM should provide a standard interface for the portability of interceptors.

2.2. Usability

Many application domains that employ an interceptor approach require a management mechanism for their interceptors. For example, Vasarhelyi et al. suggested that a CA monitoring mechanism should be easy for auditors with a limited IT background to use [34]. To be effective, the monitoring mechanism should allow auditors to work independently, i.e., as tertiary monitors [39]. Cao et al. proposed an agent-based load balance solution with strong management features to address various managerial needs [6]; while Artaiam and Senivongse use the management UI to manage the lifecycle of monitoring agents [3]. We found that the lifecycle management of the portable interceptor (PI) is an important issue for users.

We also found that, if necessary, it should be possible to reconfigure the interceptors without interrupting normal operations. For example, Whittington and Pany [39] suggested that an ideal continuous monitoring mechanism should be able to adjust in seconds without help from IT professionals [39]; while Othman et al. [26] suggested that a load balancing (LB) mechanism should be able to adapt rapidly to various load conditions [26]. Other studies also observe that there is a need to adjust the properties of an interceptor at run time to meet changing monitoring conditions [13]; and QoS Services need to adjust the parameters of the monitoring agents at run-time [3]. Therefore, we define the third system requirement:

R3 A PIM should enable administrators to manage interceptors throughout their lifecycles; in particular, a PIM should allow

[1] The major extension to [10] includes the requirement analysis via ISO/IEC 9126 and the implementation validation of the proposed PIM.

the administrators to reconfigure existing interceptors without interrupting the normal operations of any Web services running on the SOAP engine.

2.3. Reliability and efficiency

Reliability and efficiency, which are general requirements of application systems, are design and implementation issues. A number of researchers have considered these issues. For example, Artaia and Senivongse [3], Doughty et al. [8] and Rowlingson [29] emphasized that the failure of a CA monitoring mechanism should not affect the system in which it resides. Hence, our last system requirement is as follows:

R4 The operations of the interceptors and the PIM should have minimum impact on the normal operations of Web services.

2.4. Maintainability

Maintainability means "the capability of the software product to be modified [15]." Thus, a software component is maintainable if some parts of it can be reconfigured and reused. We consider that the proposed PIM is maintainable because the interceptors' lifecycles are manageable, which satisfies requirement R3.

2.5. Summary of PIM requirements

We have defined four PIM requirements based on the specifications of five application domains. Requirement R1 defines the basic functionality of the interceptors while R2 considers their portability. R3 stipulates that the PIM must be able to manage the interceptors' lifecycles, and requires that it can acquire and set the properties of the interceptor without interrupting the system's normal operations. Especially, R3 is useful in application domains when the systems have to adjust the internal parameters or the functions of the PI without interrupting any of the services of the systems. For example, a company's information system may have different business flows. If auditors want to check all the transaction flows (e.g., purchase flows and sales flows) individually at run-time, they must be able to change the auditing rules without stopping the system. Therefore, the PIM would be able to be applied to various domains, such as CA, CF, FT, LB, and QoS systems. Finally, requirement R4 relates to the impact on a modified SOAP engine if the installed PIM and/or the plugged interceptors malfunction or crash. These problems could be avoided by implementing certain technologies, which we discuss in Section 4. The above four requirements cover the six quality factors of an information system defined in the ISO/IEC 9126 standard. We believe that the proposed PIM offers sufficient functionality to build interceptors for other applications.

3. Interface design for the portable interceptor mechanism

A recognized way to achieve portability of application specific interceptors is through the use of common interfaces [30]. In Fig. 1, steps 1–4 illustrate how the PIM's administration tool (PAT) manages the interceptors' lifecycles (R3). Fig. 2 shows that exchanged messages can be intercepted at four points (R1) after interceptors have been installed in an information system. The portable interceptors, which are usually custom-made components, can be implemented by inheriting the designed interfaces. Next, we discuss the functionality offered by these two sets of interfaces through use case analysis.

3.1. Use case analysis

3.1.1. Managing an interceptor's lifecycle

To explain the management of an interceptor's lifecycle, we consider a use case in the CA domain. Suppose an auditor uses PIM's administration tool (PAT) to manage an auditing mechanism that is implemented as an interceptor, as shown in Fig. 1. The tool invokes a plug-in service call in the auditee's SOAP engine, which then finds the location (or URI) of the pluggable auditing mechanism (the interceptor) and connects to (or plugs into) the SOAP engine, as shown in the step 1 of Fig. 1. After plugging in the interceptor, the auditor sets the interceptor's initial auditing parameters, as shown in step 2, and then activates the interceptor via the same PIM administration tool, as shown in the step 3. The administrator can suspend the interceptor's functions by invoking a deactivation function in the SOAP engine, as shown in step 4. The auditor can also alter the run-time auditing options via the SOAP engine service component after the interceptor has been plugged in.

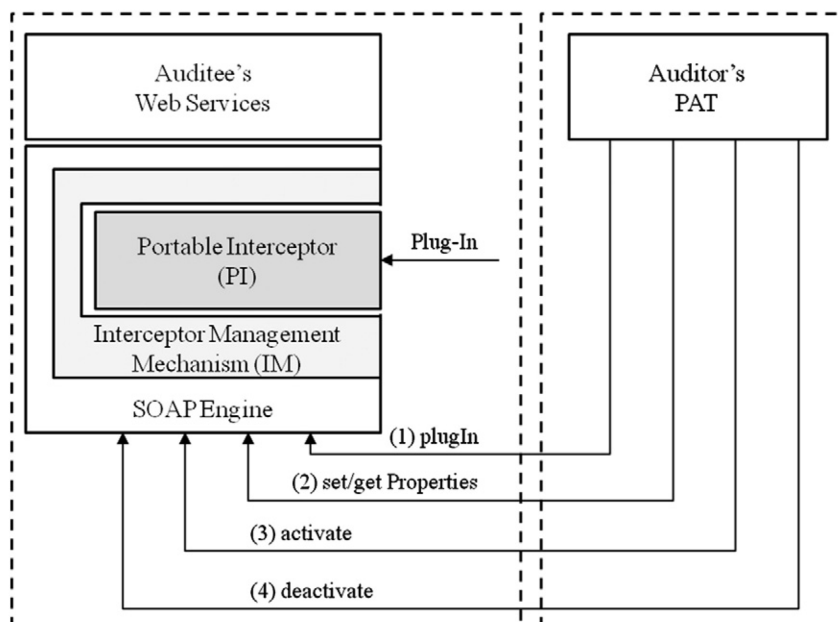


Fig. 1. Interceptor management in a Web Services server.

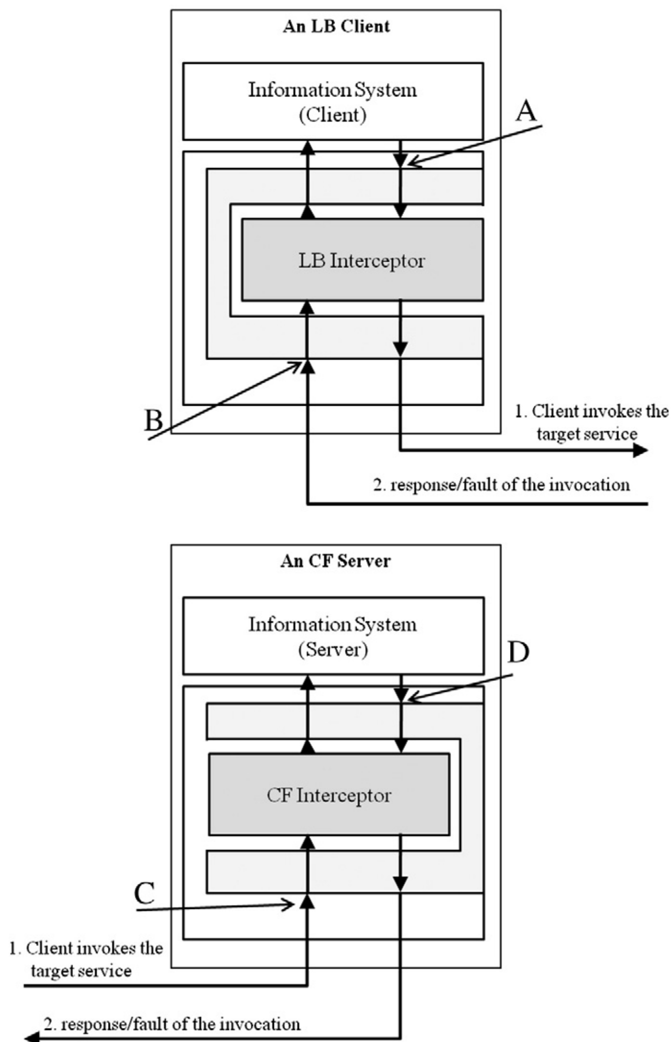


Fig. 2. The four interception points of an interceptor.

3.1.2. Run-time phase of an interceptor

An interceptor can be implemented on the server-side and/or client-side based on the application domain's need; therefore, we consider two use cases to explain its applications. Fig. 2(a) shows how a client-side interceptor is used for load balancing. The Web Service's client-side interceptor intercepts and redirects outgoing requests to the best target replica (interception point A), and processes the reply message (interception point B) when the Web Service responds. Fig. 2(b) illustrates a use case of a server-side interceptor for a CF application. The server-side CF interceptor intercepts arrival requests before the Web Service deals with the request (interception point C) and/or intercepts the response (interception point D) after the Web Service sends a reply message to the client.

2(a)An LB client-side interceptor use case 2(b)A CF server-side interceptor use case.

3.1.3. Summary of the use case analysis

The results of the use case analysis show that all the functions satisfy requirements R1–R3 of the PIM. Since the interceptor is designed to obtain SOAP messages at four points continuously, requirement R1 is fulfilled. An interceptor can be installed, uninstalled, activated, and deactivated in a SOAP engine. Furthermore, PIM has a function called *setProperties*() that allows an administrator to adjust the installed PI without interrupting the normal operations of the system; therefore, requirements R2 and R3 are also fulfilled. Requirement R4, which pertains to the system's reliability and efficiency, is considered in Section 4. Next, we discuss the design of the PIM interfaces.

3.2. Design of PIM interfaces

Based on the previous analysis, we divide the PIM interface into two application programming interfaces (API). The first is the interceptor management mechanism (IM), which is used to manage the interceptors' lifecycles at run-time. The second is the portable interceptor (PI), which ensures that the interceptors are portable and manageable by the interceptor management mechanism. We discuss the design of the management features in the SOAP engine first, followed by the design of the interceptor.

3.2.1. Design of the interceptor management mechanism API

As shown in Fig. 3, the interceptor management mechanism has five functions, which enable Web Service system administrators to manage the lifecycle of interceptors via the PIM. Specifically, the system administrator implements the administration application *admAp* to manage all interceptors via the administration APIs. The five administrative functions are: *plugIn*(), *setProperties*(), *getProperties*(), *activate*(), and *deactivate*(). An interceptor is connected to (or plugged into) the SOAP engine by using the *plugIn*(*PI_name*, *PI_module*) function. The advantage of *plugIn*() is that there are no interruptions to service during installation because the function provides a non-stop PI installation mechanism for critical Web services. The system administrator then invokes the *setProperties*(*PI_name*, *Properties*) function in the SOAP engine to set up the desired operating parameters, after which the PI is activated by calling the *activate*(*PI_name*) function in the SOAP engine. To query the PI's operating parameters, the system administrator can invoke the *getProperties*(*PI_name*) function. To deactivate the installed PI at a specific time, the administrator invokes the *deactivate*(*PI_name*) function.

To apply the PIM in different application domains, it is important to consider the data type (content) of the interface parameters. This is because the information systems in different application domains often use different data types (content) in the interface parameters to set or acquire the system's attributes. If the PIM does not apply a unified and flexible data type for the parameters, a portability problem will arise. We use a flexible data type, called Properties (written in Java language) for the interface parameters. As Properties (shown in line 5 of Fig. 7) can be used to transfer various types of data, our PIM can be applied in different application domains.

Fig. 4 shows an excerpt from the interceptor manager interface in WSDL [37] format; however, the format is not really suitable for presentation purposes. Fortunately, OMG provides a mapping specification between OMG's IDL and WSDL [24], which allows us to convert the PI manager into OMG IDL format. An excerpt from the interceptor manager's interface in IDL format is shown Fig. 5. Hereafter, we use OMG IDL to present the interface.

3.2.2. Design of the interceptor API

To design the interceptor API, we consider the results of the use case analysis. Messages can be intercepted at four points in an information system. Our interceptor interface is comprised of a basic interceptor, a client-side interceptor, and a server-side interceptor, as shown in Fig. 6. The basic interceptor interface, called the **Interceptor**, is described in Lines 2 to 10 of Fig. 7. After the interceptor has been loaded in the initialization phase, the SOAP engine calls the *Interceptor*::*initialize*() function (Line 3). The interceptor programmers may set up the necessary running environment parameters for all desired purposes. The *Interceptor*::*destroy*() function (Line 4) allows the SOAP engine to clean up an unused interceptor for management purposes in the terminal phase. The *Interceptor*::*setPropoerties*(*PI_name*,*Properties*) and *Interceptor*::*getPropoerties*(*PI_name*) functions allow the administrator to set additional properties and query the parameters' properties at

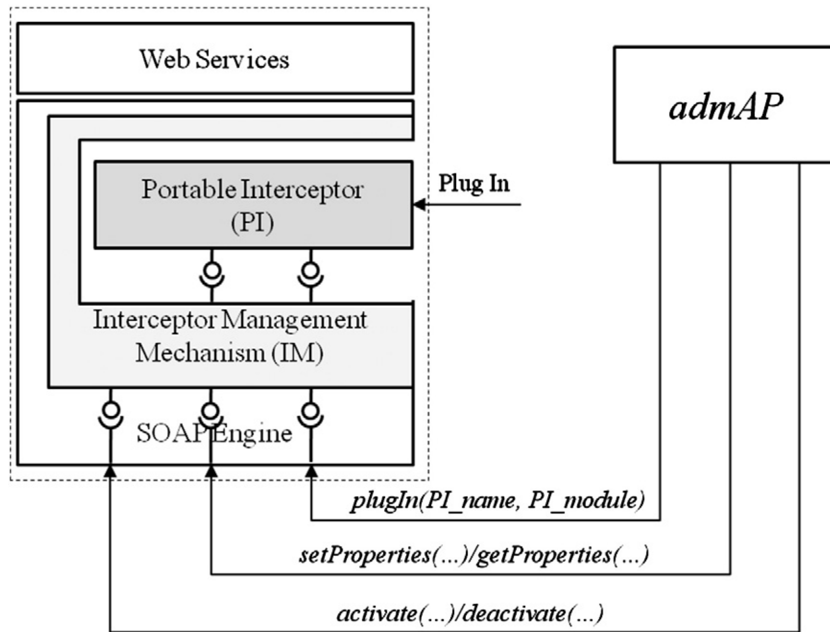


Fig. 3. An example of the interceptor management module.

run-time. When the *IIManagerImpl*::*activate*() function is invoked, the SOAP engine calls *Interceptor*::*activate*() to activate the interceptor; and when the *IIManagerImpl*::*deactivate*() function is invoked, the engine calls *Interceptor*::*deactivate*() to temporarily deactivate the interceptor. The administration application performs these operations at the request of the administrator.

Since the basic interceptor is inherited by the **ClientInterceptor** and the **ServerInterceptor**, they also have the above basic features. The **ClientInterceptor** is defined in Lines 16–19 of Fig. 7. The *ClientInterceptor*::*sendRequest*(*RequestInfo*) function is invoked by the SOAP engine before a client request is sent to the server. The interceptor's programmers can implement their desired intercepting code via this function. The *ClientInterceptor*::*receiveReply*(*RequestInfo*) function is invoked by the SOAP engine after it receives a response from the server so that the interceptor can check the response. The proposed **ServerInterceptor**, defined in Lines 21–24 of Fig. 7, determines the two server-side interception points *ServerInterceptor*::*receiveRequest*() and *ServerInterceptor*::*sendReply*(). The operation of this interface is similar to that of **ClientInterceptor**.

4. Implementing the concept

We analyze the feasibility of applying the proposed PIM via a proof of concept. Specifically, we implement two prototypes and evaluate them based on the determined requirements. We share the experience we gained from implementing the PIM prototype on two open-source SOAP frameworks: Apache Axis 1.x [1] and Codehaus XFire [7]. Then, we test the operability (requirements R1, R2, and R3) of the PIM and use the implemented prototypes to evaluate

```
<?xml version="1.0" ?>
<definitions name="IIManagerImpl" ...>
  <types>
    <schema ...>
      <xsd:simpleType name="PI_module">...
      <xsd:simpleType name="PI_name"> ...
    </schema>
  </types>
  <message name="plugIn">
    <part name="PI" type="xsd:PI_name" />
    <part name="pim" type="xsd:PI_module" />
  </message>...
  <message name="setPropertiess">
    <part name="props" type="xsd:Properties" />
  </message>...
  <message name="getProperties">
  </message>...
  <message name="activate">
    <part name="PI" type="xsd:PI_name" />
  </message>...
  <message name="deactivate">
    <part name="PI" type="xsd:PI_name" />
  </message>...
  <portType name="IIManagerImplPortType">
    <operation name="plugIn">...
    <operation name="setPropertiess">...
    <operation name="getProperties">...
    <operation name="activate">...
    <operation name="deactivate">...
  </portType>
  <binding ...>
.....
  </binding>
  <service name="IIManagerImplService">
    ...
  </service>
</definitions>
```

Fig. 4. An excerpt from the interceptor manager's interface in WSDL format.

```
typedef string PI_name;
typedef string PI_module;

interface IIManager {
  void plugIn(in PI_name PI, in PI_module pim);
  void setProperties(in PI_name PI, in Properties props);
  Properties getProperties(in PI_name PI);
  boolean activate(in PI_name PI);
  boolean deactivate(in PI_name PI);
}
```

Fig. 5. An excerpt from the interceptor manager's interface in OMG IDL format.

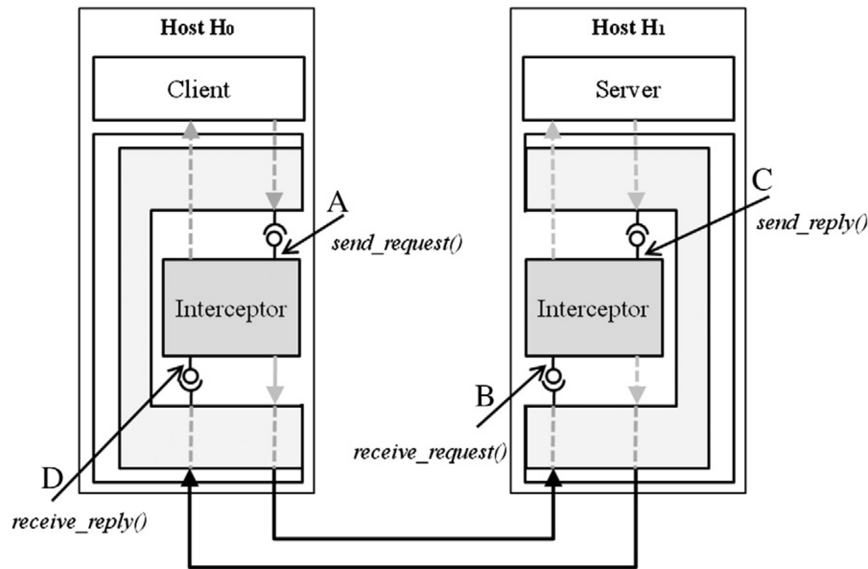


Fig. 6. The API design for the interceptor.

requirement R4. The implementations and the experimental results show that the prototypes meet all four requirements of the PIM.

4.1. Prototype implementation

There are many SOAP frameworks, which have similar architecture, such as Axis [1], XFire [7], JBoss [18], and so on. We build the PIM prototypes on Apache Axis 1.x [1] and Codehaus XFire [7] since they are widely used in academic research, for example, grid services [5,35], sharing digital resources [33], and composite telecom services [42]. Each framework provides a handler mechanism [1,7] that allows application developers to combine their handlers as a handler chain. Note that the handlers are similar to the interceptors in our context. In both frameworks, when the SOAP engines redirect a message to the handler chain, it passes through each handler; hence, each handler has an opportunity to examine the message content, as stipulated in requirement R1. Because of the proprietary nature of handler mechanisms, the handlers (or interceptors), developed for Apache and XFire are not portable to the other frameworks, even though the frameworks' handler mechanisms are very similar. Each handler mechanism provides "handler management" functions that enable application developers to add, delete, or modify the handler chain through the configuration files; however, the SOAP engine has to be restarted each time a modification is made, which means requirement R3 is not satisfied completely. Finally, neither framework provides a way to address the reliability issue, as called for in requirement R4; thus, a design fault or software failure in a handler could cause the whole system to crash. Next, we describe the implementation of PIM on both frameworks to demonstrate its feasibility.

The PIM interface comprises the client-side interfaces and the server-side interfaces. Here, we only consider the server-side implementation of PIM, since the client-side implementation is similar. In Apache Axis, the functionalities of message exchange between the handlers and the SOAP engine, as well as handler management are implemented in one class, namely **org.apache.axis.server.AxisServer.java**. Therefore, we modify the source file to embed the interceptor management mechanism (IM) for managing the lifecycle of interceptors. In contrast, XFire separates bidirectional message exchanges and handler management into different classes. Hence, we create two new handlers (two class files), called **org.codehaus.xfire.handler.RequestPIMHandler.java** and **org.codehaus.xfire.handler.ResponsePIMHandler.java**, as an IM. Next, we modify two source files, **org.codehaus.xfire.handler.Phase.java** and **org.codehaus.xfire.DefaultXFire.java**, to register and enable the IM, which is then installed on the SOAP engine of XFire. Fig. 8 shows an excerpt from the *IIManager's* interface in Java, and Fig. 9 shows a possible implementation of *IIManager.* The interface is converted directly from the IDL definition of *IIManager*. Lines 7 and 8 in Fig. 9, show how to load and manage the given interceptor via the *IIManagerImpl::plugIn*() function; and lines 11 and 14 configure the attributes of the interceptor.

PIM defines a set of abstract interfaces via which an interceptor can be incorporated with the SOAP engine and process SOAP

```
1 : //IDL definitions of the interceptor
2 : local interface Interceptor {
3 :     void initialize();
4 :     void destroy();
5 :     void setProperties(in PI  name PI, in Properties props);
6 :     Properties getProperties(in PI  name PI);
7 :     void activate();
8 :     void deactivate();
9 :       …
10 : };
11 : local interface RequestInfo {
12:     string target;
13:     string operation;
14 :   …
15 : };
16 : local interface ClientInterceptor : Interceptor {
17:     void sendRequest (in RequestInfo ri);
18:     void receiveReply (in ReplyInfo ri);
19: };
20:
21: local interface ServerInterceptor : Interceptor {
22:     void receiveRequest (in RequestInfo ri);
23      void sendReply (in ReplyInfo ri);
24 };
```

Fig. 7. An excerpt from the interceptor manager's interface in OMG IDL format.

```
package org.w3c.PortableInterceptor;
public interface IIManager extends java.rmi.Remote{
   void plugIn(PI_name PI, PI_module pim);
   void setProperties(PI_name PI, Properties props);
   Properties getProperties(PI_name PI);
   boolean activate(PI_name PI);
   boolean deactivate(PI_name PI);
}
```

Fig. 8. An excerpt from the IIManager's interface.

```
1 : package org.apache.axis.PIM;
2 : ...
3 : public class IIManagerImpl implements IIManager {
4 :     private Hashtable plugedInterceptor = new HashTable ;
5 :     void plugIn(PI_name PI, PI_module pim){
6 :       ...
7 :       ServerInterceptor si=Class.forName(pim);
8 :       plugedInterceptor.put(PI, si);
9 :       ..
10:    };
11:    void setProperties(PI_name PI, Properties props){
12:       ...
13:    };
14:    Properties getProperties(PI_name PI){
15:       ...
16:    };
17:    boolean activate(PI_name PI){
18:       ...
19:    };
20:    boolean deactivate(PI_name PI){
21:       ...
22:    };
23 :}
```

Fig. 9. An excerpt from the IIManager's implementation code for Apache Axis.

messages. Next, we explain the implementation of a server-side interceptor; a client-side interceptor can be implemented in similar manner. Figs. 10, 11, and 12 show excerpts from the interfaces converted directly from the related IDL in Fig. 7. After interceptor developers implement the interfaces based on their specific requirements, they can use *IIManagerImpl* to load and manage their interceptors. Here, we implement a server-side interceptor as a later "plug-in" for the two IMs implemented on the respective frameworks. The server-side interceptor utilizes the functions *receiveRequest(…)* and *sendReply(…)* from the **org.w3c.PortableInterceptor.ServerInterceptor** interface (shown in Fig. 12). The client-side interceptor implements the functions *sendRequest(…)* and *receiveReply(…)* from the **org.w3c.PortableInterceptor.ClientInterceptor** interface (shown in Fig. 11).

During the implementation period, we use *exception handling* to improve the reliability (R4) of the SOAP engine when the PIM is installed because, in general, the system will be more reliable if all its exceptions have been handled completely. In our experience, when an application system implements a PIM, malfunctions of the IM and/or interceptors have the greatest impact on the reliability of the system. To prevent such situations, we add programming codes in the SOAP engine to catch and mask all exceptions that may occur at the IM. In the same way, we can add programming codes to the IM module to catch and mask all exceptions that may occur at the interceptors. Since different SOAP engine providers may use different architecture and/or implementation, developers have a major challenge while they implement a PIM compliant SOAP engine. Because of developers have to hack each source code of the SOAP engines and figure out their entry point of SOAP messages to apply the proposed PIM. However, we think that SOAP framework providers could easily apply the proposed PIM to their products.

```
package org.w3c.PortableInterceptor;
/* All potable interceptors implement this interface
public interface Interceptor
{
   void initialize() ;
   void destroy() ;
   void activate() ;
   void deactivate() ;
   ...
} // interface Interceptor
```

Fig. 10. An excerpt from the interceptor interface definition in Java.

```
package org.w3c.PortableInterceptor;
public interface ClientInterceptor extends Interceptor
{
   void sendRequest(...) ;
   void receiveReply(...) ;
} // interface ClientInterceptor
```

Fig. 11. An excerpt from the ClientInterceptor interface definition in Java.

4.2. Experiments on requirements R4 of PIM

In this subsection, we evaluate the implemented prototypes based on the requirements of PIM. Fig. 13 shows the test environment. Hosts $\mathbf{H_0}$, $\mathbf{H_1}$, and $\mathbf{H_2}$ use a Pentium IV 3.2GHz PC with a 1 GB memory and run on an MS Windows XP platform. An administration application (*admAP*) and a client are implemented in MS C# and installed on Host $\mathbf{H_0}$. Meanwhile, hosts $\mathbf{H_1}$ and $\mathbf{H_2}$ are installed, respectively, in Axis and XFire, the modified open source SOAP engines. First, we consider the operability of the prototypes (requirements R1, R2, and R3), and then evaluate requirement R4.

Here, we test the operability (the run-time lifecycle management) of the IMs and the portability of the implemented interceptor (PI), which can be used in both Axis and XFire. To test the operability of the IMs, the administration application first invokes the *plugin*() and *activate*() functions to plug in and activate our test interceptor (Steps 1 to 4). Next, the client invokes a Web service, such as the *invoke*() function, provided by both SOAP engines (Steps 5 to 6). The test results show that both IMs can manage the implemented interceptor (PI) at run-time and that (PI) can be used by different SOAP engines. We demonstrate that messages can be intercepted by developing a SOAP client (shown in Fig. 14), which sends a simple sales order to the Web Services $\mathbf{H_1}$ and $\mathbf{H_2}$ and receives a result marked "true". The intercepted messages (shown in Figs. 15 and 16) can be used to monitor and analyze the contexts in real-time. The SOAP messages dumped by the Axis and XFire engine consoles show that the server-side interceptor can function on both SOAP engines.

Requirement R4 of PIM relates to the impact on the modified SOAP engine if the installed IM and/or the plugged interceptors malfunction or crash. Recall that we discussed the reliability of PIM in Subsection 4.1; here, we describe the experiment for evaluating its efficiency. The PIM's overhead is incurred primarily by the IM and the interceptors. The IM increases the run-time overhead when it looks up the interceptor chain (interceptors) and calls the interception functions of the interceptor. The overhead of an interceptor depends on its internal algorithms and activities. However, interceptors are usually custom-made, so it is difficult to determine the performance of each one. We therefore evaluate the overhead of a PIM that includes an IM and a dummy interceptor. Fig. 17 shows an excerpt from the dummy interceptor's code. The interceptor can also be used to dump SOAP messages when the interception function *receiveRequest*() is called (Line 6). We construct an experiment environment that is similar to the one in Fig. 13; however, we place $\mathbf{H_0}$, $\mathbf{H_1}$, and $\mathbf{H_2}$ on the same PC to eliminate network latency. The client separately invokes the *echoString*() belonging to the Web Services deployed on the modified Axis and XFire SOAP engines. The purpose of the experiment is to measure the response time delay incurred by the working PIM. All the results reported in this section have a 95%

```
package org.w3c.PortableInterceptor;
public interface ServerInterceptor extends Interceptor
{
   void receiveRequest(...) ;
   void sendReply(...) ;
} // interface ServerInterceptor
```

Fig. 12. An excerpt from the ServerInterceptor interface definition in Java.

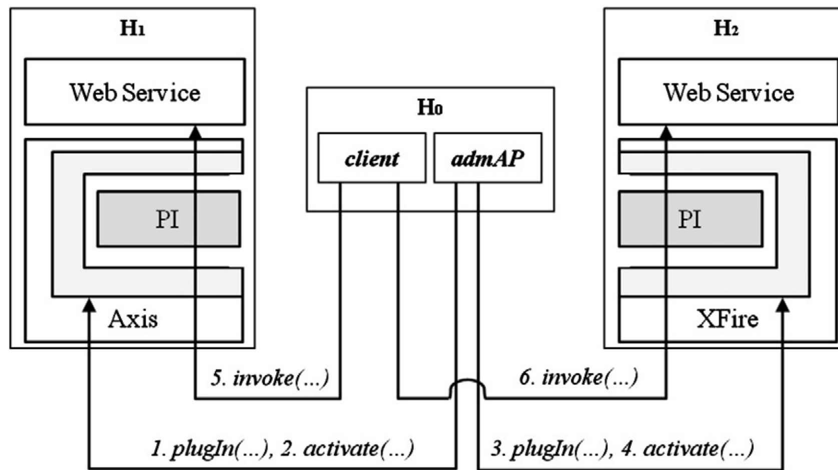


Fig. 13. The test environment for evaluating the portability of the proposed PIM.

Fig. 14. An example of a SOAP client.

confidence interval with interval half-widths of less than 3% of the average measurements.

Analysis of the overhead shows that the size of the input/output parameter (or the message size) in each nested invocation can affect the invocation's round trip time (RTT). Therefore, we measure the PI's net processing time for messages that vary in size from 4 bytes to 500 KB. The experiment results, shown in Table 1, demonstrate that the client's RTT increases as the message size increases. The reduction in the RTT with PIM on XFire for 500 KB of data may be due to the external operation of the system I/O. Since PIM's overhead is less than that of the system I/O, we conclude that the PIM overheads on Axis and XFire are negligible. Thus, based on the proof of concept previously discussed, we believe that PIM is feasible.

5. Related works

There are two commonly used interception mechanisms, namely the SOAP intermediary mechanism [38] and the CORBA portable interceptor (PI) [25]. In this section, we discuss the feasibility of using the SOAP intermediary and the CORBA portable interceptor to satisfy the four PIM requirements.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tns="http://localhost/axis/services/OrderRecvImpl"
    xmlns:types="http://localhost/axis/services/OrderRecvImpl/encodedTypes"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<q1:OrderRecv xmlns:q1="http://DefaultNamespace">
<strOrderID xsi:type="xsd:string">960101000001</strOrderID>
<strCompanyName xsi:type="xsd:string">DMS Corporation</strCompanyName>
<strProductName xsi:type="xsd:string">Double A</strProductName>
<lngTotalNo xsi:type="xsd:long">100</lngTotalNo>
<lngPrice xsi:type="xsd:long">100</lngPrice>
<lngTotalAmount xsi:type="xsd:long">10000</lngTotalAmount>
</q1:OrderRecv>
</soap:Body>
</soap:Envelope>
```

Fig. 15. A message intercepted at the interception point of *receiveRequest(…)*.

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<ns1:OrderRecvResponse
    soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:ns1="http://DefaultNamespace">
<OrderRecvReturn xsi:type="xsd:boolean">true</OrderRecvReturn>
</ns1:OrderRecvResponse>
</soapenv:Body>
</soapenv:Envelope>
```

Fig. 16. A message intercepted at the interception point of *sendReply(…)*.

5.1. The flexibility of using the SOAP intermediary mechanism to meet PIM requirements

The W3C recommendation for SOAP 1.2 proposes an intermediary mechanism [38], which operates as a proxy Web service that receives SOAP messages redirected from either the client site (invocation messages) or the server site (reply messages). The redirected information is inserted into the header of a SOAP message before the latter is sent out. An intermediary can examine the content of a SOAP message and process it in the same way as an interceptor in the PIM. Note that an intermediary operates as a third party Web service independent of the client application and the server site's Web Service. In other words, its role is different to that of the client site interceptor and the server site interceptor (the endpoint interceptor), as requirement R1, so the intermediary cannot implement some of the interceptor's applications. For example, a popular QoS metric is used to measure the round-trip delay of a SOAP invocation; however, it is difficult to obtain a precise measurement using the intermediary approach because the intermediary site is located between the client site and the server site. Besides, a fault tolerant system needs an endpoint interceptor to log and to recover states of the system; however, an intermediary is only operated as a midpoint proxy. Since an intermediary does not quite meet the functional requirement R1, we do not have to discuss the requirements R2–R4.

5.2. The flexibility of using CORBA's portable interceptor to meet PIM requirements

CORBA [23] provides a “Portable Interceptor” [25] recommendation function in its standard specification. The recommendation function defines a set of interfaces that enable an application-specific interceptor to communicate with the CORBA ORB in order to intercept the content of the latter's in/out invocations (R1). The information in a CORBA invocation is equivalent to that of a SOAP message. The recommendation function also allows software developers to install interceptors that are portable (R2) across all CORBA 3.0 compliant middleware. In addition, software developers can use the interfaces to manage some aspects of their interceptors, but the interfaces are not adaptable, as stipulated in R3. Because of CORBA ORB has to be restarted for applying new configuration while the CORBA-based system has been reconfigured. However, some CORBA vendors, such as VisiBroker [4] and OpenORB [40], provide proprietary solutions that partially meet requirement R3. Although the CORBA standard does not specifically address the reliability issue in R4, we found that state-of-the-art CORBA compliant products, such as Orbix [16] and OpenORB [40], satisfy R4 because they use proprietary implementation techniques.

```
1:  package tw.edu.ntou.cs.dms.PortableInterceptor;
2:  import javax.xml.soap.SOAPMessage;
3:  import org.w3c.PortableInterceptor.ServerInterceptor;
4:  public class ServerPI01 implements ServerInterceptor {
5:  public void receiveRequest(SOAPMessage reqMsg) {
6:          //System.out.println(reqMsg.toString());
7:  }
...
 }
}
```

Fig. 17. An excerpt from the dummy service interceptor code.

6. Conclusion

Simple Object Access Protocol (SOAP) frameworks are commonly used platforms for developing Web Service, which constitute an increasingly recognized technology for constructing distributed applications. An interceptor is generally applied to intercept and monitor messages for specific purposes. We believe that Web Services would have greater applicability in various application domains if SOAP frameworks were to incorporate an interceptor mechanism. To test this proposition, we first acquired the requirements of an interceptor mechanism by examining how various application domains, including continuous auditing (CA), computer forensics (CF), fault tolerance (FT), load balancing (LB), and quality of service (QoS), employ the interceptor approach. Second, to satisfy these requirements, we design a set of standard interfaces, called “portable interceptor mechanism” (PIM), for SOAP engines (which are core component of SOAP frameworks). Finally, we evaluated the feasibility of applying the PIM by using a proof of concept that implements two prototypes and examines them based on the determined requirements. The results show that the proposed PIM is feasible.

Table 1
Experimental data for client RTT delay.

Data size	4 bytes	1 KB	2 KB	5 KB	50 KB	500 KB
(A) Axis						
RTT with PIM (ms)	1.64	1.94	2.18	3.08	14.53	116.90
RTT w/o PIM (ms)	1.62	1.89	2.13	3.01	14.37	116.61
PIM overhead	0.02	0.04	0.05	0.07	0.15	0.29
(B) XFire						
RTT with PIM (ms)	1.10	1.11	1.19	1.53	6.18	98.55
RTT w/o PIM (ms)	1.09	1.11	1.19	1.52	6.18	99.01
PIM overhead	0.01	0.00	0.00	0.01	0.00	−0.46

The proposed PIM provides three major features: endpoint interception, portability, and run-time lifecycle management. These features are not fully supported by existing solutions. First, the proposed PIM supports a set of standard interfaces for endpoint interception, which differs from the SOAP intermediary that operates as a midpoint proxy. An endpoint interception is required by some application domains such as FT, QoS, and CF. Second, the interceptors can become portable across systems running on SOAP frameworks when both interceptors and frameworks are compliant with the proposed PIM. This is in contrast to the proprietary implementation of existing SOAP frameworks such as the Axis's "handler." Third, the proposed PIM provides an interface for managing the interceptors' lifecycle at run-time, which is a crucial functionality for applications such as CA and LB. This essential feature is not fully supported by existing solutions.

References

[1] Apache, Apache Axis 1.x Documents, from http://ws.apache.org/axis/.
[2] P. Arpaia, M. Bernardi, G. Lucca, V. Inglese, G. Spiezia, An Aspect-Oriented Programming-based approach to software development for fault detection in measurement systems, Computer Standards & Interfaces 32 (2010) 141–152.
[3] N. Artaiam, T. Senivongse, Enhancing Service-Side QoS Monitoring for Web Services, Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008, pp. 765–770.
[4] Borland, VisiBroker 7.0 Manual, from http://info.borland.com/techpubs/visibroker/ 2006.
[5] K. Benedyczak, A. Nowinski, K.S. Nowinski, P. Bala, UniGrids Streaming Framework: Enabling Streaming for the New Generation of Grids, Lecture Notes in Computer Science (Applied Parallel Computing, State of the Art in Scientific Computing 4699 (2008) 809–818.
[6] J. Cao, D. Spooner, S. Jarvis, G. Nudd, Grid load balancing using intelligent agents, Future Generation Computer Systems 21 (1) (2005) 135–149.
[7] Codehaus, XFire user's guide (XFire 1.0 Document), from http://xfire.codehaus.org/ 2006.
[8] K. Doughty, J. O'Driscoll, Information technology auditing and facilitated control self-assurance, Information Systems Control Journal 4 (2002) 33–38.
[9] C.L. Fang, D. Liang, C. Chen, P. Lin, A Redundant Nested Invocation Suppression Mechanism for Active Replication Fault Tolerant Web Service, Proc. of the IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'04), Taipei, March 2004, pp. 9–16.
[10] C.L. Fang, D. Liang, F. Lin, C.C. Lin, W. Chu, A Portable Interceptor Mechanism on SOAP for Continuous Audit, Proc. of 13th Asia-Pacific Software Engineering Conference (APSEC2006), Bangalore, India, December 2006, pp. 95–104.
[11] P. Felber, P. Narasimhan, Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems, IEEE Transactions on Computers 53 (5) (2004) 497–511.
[12] M. Gunestas, D. Wijesekera, A. Singhal, Forensic Web Services, IFIP International Federation for Information Processing 285 (2008) 163–176.
[13] J. Hallstrom, W. Leal, A. Arora, Scalable evolution of highly available systems, IEICE Transactions on Information and Systems 86 (10) (2003) 2154–2164.
[14] A. Helali, A. Soudani, S. Nasri, T. Divoux, An approach for end-to-end QoS and network resources management, Computer Standards & Interfaces 28 (2005) 93–108.
[15] ISO/IEC, 9126, Software engineering — Product quality, from http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749.
[16] IONA, Technologies, Orbix 3.3, from http://www.iona.com/products/orbix/.
[17] N. Jailani, N. Yatim, Y. Yahya, A. Patel, M. Othman, Secure and auditable agent-based e-marketplace framework for mobile users, Computer Standards & Interfaces 30 (2008) 237–252.
[18] JBoss, JBoss by Red Hat, http://www.jboss.com/.
[19] A. Kogan, E. Sudit, M. Vasarhelyi, Continuous online auditing: a program of research, Journal of Information Systems 13 (2) (Fall 1999) 87–103.
[20] D. Liang, C. Fang, C. Chen, F. Lin, Fault Tolerant Web Service, *Proc. of 10th Asia-Pacific Software Engineering Conference* (APSEC2003), Chiang Mai, Thailand, December 2003, pp. 310–321.
[21] P. Narasimhan, M. Moser, P. Melliar-Smith, Strongly consistent replication and recovery of fault-tolerant CORBA applications, Computer System Science and Engineering Journal 17 (2) (March 2002) 103–114.
[22] B. Natarajan, A. Gokhale, S. Yajnik, D. Schmidt, DOORS: Towards High-Performance Fault Tolerant CORBA, The 2nd Distributed Applications and Objects (DOA) conference, 2002, pp. 39–48.
[23] Object Management Group (OMG), Common Object Request Broker Architecture: Core Specification V 3.0.3, OMG Technical Committee Document formal/04-03-01, 2004.
[24] Object Management Group (OMG), WSDL-SOAP to CORBA Interworking, OMG Technical Committee Document mars/03-05-07, 2003.
[25] Object Management Group (OMG), CORBA Portable Interceptors V 3.0.3, OMG Technical Committee Document formal/04-03-19, 2004.
[26] O. Othman, J. Balasubramanian, D. Schmidt, The Design of an Adaptive Middleware Load Balancing and Monitoring Service, LNCS/LNAI: Proceedings of the Third International Workshop on Self-Adaptive Software, Heidelberg, June 2003.
[27] A. Patel, Frameworks for secure, forensically safe and auditable applications, Computer Standards & Interfaces 30 (2008) 213–215.
[28] Z. Rezaee, A. Sharbatoghlie, R. Elam, Continuous auditing: building automated auditing capability, Auditing: A Journal of Practice & Theory 21 (1) (March 2002) 147–163.
[29] R. Rowlingson, A ten step process for forensic readiness, International Journal of Digital Evidence 2 (3) (Winter 2004).
[30] J. Snell, T. Glover, Portability and interoperability, from http://www.ibm.com/developerworks/webservices/library/ws-port/index.html?S_TACT=105AGX04&S_CMP=EDU 2003.
[31] D. Song, dSniff 2.3 Document, from http://www.monkey.org/dugsong/dsniff/ 2001.
[32] P. Sommer, Intrusion detection systems as evidence, Computer Networks 31 (23–24) (1999) 2477–2487.
[33] P. Toukach, H.J. Joshi, R. Ranzinger, Y. Knirel, C.-W. Lieth, Sharing of worldwide distributed carbohydrate-related digital resources: online connection of the Bacterial Carbohydrate Structure DataBase and GLYCOSCIENCES, de, Nucleic Acids Research (Database issue) 35 (2008) 280–286.
[34] M. Vasarhelyi, F. Halper, The continuous audit of online systems, Auditing: A Journal of Practice and Theory 10 (1) (1991) 110–125.
[35] V. Venturi, M. Riedel, S. Memon, F. Stagni, B. Schuller, D. Mallmann, B. Tweddell, A. Gianoli, S. Berghe, D. Snelling, A. Streit, Using SAML-Based VOMS for Authorization within Web Services-Based UNICORE Grids, Lecture Notes, Computer Science (Euro-Par 2007 Workshops: Parallel Processing 4854 (2008) 112–120.
[36] W3C, Simple Object Access Protocol (SOAP) 1.1 recommendation, from http://www.w3.org/TR/SOAP/ 2000.
[37] W3C, Web Services Description Language (WSDL) 1.1, from http://www.w3.org/TR/wsdl 2001.
[38] W3C, Simple Object Access Protocol (SOAP) 1.2 recommendation, from http://www.w3.org/TR/soap12-part1/ 2003.
[39] R. Whittington, K. Pany, Principles of Auditing and other Assurance Services, McGraw-Hill, 2002.
[40] C. Wood, J. Daniels, M. Rumpf, OpenORB v.1.4 manual, from http://openorb.sourceforge.net/docs/1.4.0/OpenORB/doc/orb.html 2004.
[41] J. Woodroof, D. Searcy, Continuous audit model development and implementation within a debt covenant compliance domain, International Journal of Accounting Information System 2 (2001) 169–191.
[42] Y. Yuan, J.J. Wen, W. Li, B.B. Zhang, A Comparison of Three Programming Models for Telecom Service Composition, The Third Advanced International Conference on Telecommunications (AICT'07), 2007, p. 1.
[43] L. Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, H. Chang, QoS-Aware Middleware for Web Services Composition, IEEE Transactions on Software Engineering 30 (5) (2004) 311–327.

Chien-Cheng Lin is currently attending the National Taiwan Ocean University, Taiwan, pursuing a PhD in computer science. He also earned his MS in computer science there in 2003. His research interests include fault-tolerance, information security, and digital forensics.

Chen-Liang Fang is an associate professor at Department of Information Management, Jinwen University of Science and Technology, Taiwan. He received his PhD degree from National Taiwan University of Science and Technology, Taiwan. His research interests include fault-tolerance, and information security.

Deron Liang received a BS degree in electrical engineering from National Taiwan University in 1983, and an MS and a PhD in computer science from the University of Maryland at College Park in 1991 and 1992 respectively. He is on the faculty of Computer Science & Information Engineering Department, and serves as Director of Software Research Center, National Central University, Taiwan since 2008. Dr. Liang's current research interests are in the areas of software fault-tolerance, system security, and system reliability analysis. Dr. Liang is a member of ACM and IEEE.