# A mobile agents-based approach to test the reliability of web services

Jia Zhang

Department of Computer Science
Northern Illinois University
DeKalb, IL 60115, USA
E-mail: jiazhang@cs.niu.edu

**Abstract:** The paradigm of web services has been transforming the internet from a repository of data into a repository of services, or so-called web services. As more and more web services are published on the internet, how to opt for an appropriate and trustworthy web service poses a big challenge. In this paper we propose a mobile agents-based approach that selects reliable web service components in a cost-effective manner.

**Keywords:** web services; software reliability; mobile agents.

**Reference** to this paper should be made as follows: Zhang, J. (2006) 'A mobile agents-based approach to test the reliability of web services', *Int. J. Web and Grid Services*, Vol. 2, No. 1, pp.92–117.

**Biographical notes:** Jia Zhang is an Assistant Professor of the Department of Computer Science at Northern Illinois University. She is now with BEA Systems Inc. She is also a Guest Scientist at the National Institute of Standards and Technology (NIST). Her current research interests centre around software trustworthiness in the domain of web services, with focus on reliability, integrity, security, and interoperability. Zhang received her PhD in Computer Science from the University of Illinois at Chicago in 2000. She is member of the IEEE and ACM.

This article is a significant extension to the author's paper 'An approach to facilitate reliability testing of web service components' that appeared in the *Proceedings of IEEE 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, 2–5 November 2004, Saint-Malo, Bretagne, France, pp.210–218.

## 1 Introduction

Simply put, a web service refers to a programmable web application that is universally accessible using standard internet protocols (Ferris and Farrell, 2003). This paradigm of web services has been changing the face of internet from a repository of data into a repository of services in the following three ways:

1 In a way that business organisations make their software services accessible on the internet through standard programmatic interfaces, this model of web services facilitates Business-to-Business (B2B) e-Commerce within and across organisational boundaries (Fremantle *et al*., 2002).

2 The web services technology provides a uniform framework to increase cross-language and cross-platform interoperability for distributed computing and resource sharing over the internet.

3 The paradigm of web services represents a cost-effective way to engineer software by quickly composing and integrating existing web applications to conduct new business transactions.

These three capabilities are powerfully shaping internet computing for the future. Therefore, the paradigm of web services is considered to be the strategic model for the next generation of internet computing (Holland, 2002; Stal, 2002). Indeed, Gartner Group, a leading industry analyst firm, predicted that by 2008 more than 60% of business will adopt web services and transform these into new types of enterprises (Pezzini, 2003).

However, not everyone is willing to take the web services plunge. TechMetrix's survey data revealed that the actual adoption of web services in industry was quite slow: up to February 2003, although most of the companies showed interest in the idea of using web services, only 26 percent of the surveyed enterprises started web services projects. Moreover, most of those companies offered web services instead of actually consuming them (TechMetrix/SQLI 2003). Although web services are a boom to e-Commerce, most of the business stakeholders do not have enough trust on the current developed web services.

It is not clear that this new model of web services provides any measurable increase in software trustworthiness (Parnas *et al*., 1990), which can be assessed by the set of classic software 'ilities,' such as reliability, scalability, efficiency, security, usability, adaptability, maintainability, availability, portability, *etc*. (Neumann, 2004). The essential feature of 'dynamic discovery and integration' of web services model, among other aspects, poses new challenges to software trustworthiness. In a traditional software system, all contained components and their relationships are pre-decided before the software runs. Therefore, each component can be thoroughly tested, and the interactions among the components fully examined, before the system starts to execute. Web services extend this paradigm by providing a more flexible approach to dynamically locate and assemble distributed web services in an internet-scale setting. In more detail, when a system requires a web service component, it will search a public registry where web services providers publish their services, choose the optimal web service that fulfils the requirements, bind to the selected service's website, and invoke the web service. In other words, in this dynamic invocation model, it is likely that users may not even know which web services will be used until run-time (Gold *et al*., 2004), much less those web services' trustworthiness (*i.e.*, their 'ilities'). Consequently, how to select qualified web services remains a challenge. Even worse, since web services are hosted by their own providers over the internet, remotely testing the trustworthiness of web services via constant binding is neither efficient nor effective. Furthermore, how to test the interoperability of a remote web service in a specific software environment remains another challenge.

In summary, the flexibility of web services-oriented computing is not without penalty since the value added by this new paradigm can be largely defeated if:

1 the selected web service components do not thoroughly fulfil the requirements (*i.e.*, functionally or nonfunctionally)

2 the hosts of web service components act maliciously or errantly at invocation times

3 erratic internet behaviours or resource scarcity pose unendurable time delays

4 the selected web service components act errantly in the composed environment.

Meanwhile, web services technology is still in its infancy, and the trustworthiness has not gained significant attention. To date, research is preoccupied with low-level technical mechanisms of web services, *e.g.*, how to publish a web service, how to compose web services, what is overall architecture of web services-oriented system, *etc*. As web services are assimilated more into the mainstream, web services trustworthiness will hinder the adoption of web services. Therefore, how to measure, test, and enhance the trustworthiness of loosely coupled web services-oriented systems requires immediate attention.

In practice, a fundamental question needs to be answered first: Is the trustworthiness of web services really measurable and testable? Our position is that, since general software trustworthiness can be viewed as containing a number of 'ilities' (Neumann, 2004), if each 'ility' of a web service-oriented system is measurable and testable, then the trustworthiness of the whole system is measurable and testable. In other words, if a web service-oriented system scores high in every 'ility', (Here we omit the exception that some ilities naturally conflict with each other, such as fault tolerance and testability.) it is safe to say that the trustworthiness of the system is high.

Therefore, a feasible strategy is to first independently investigate each individual ility in the domain of web services before exploring the mixed ilities. As the first step, in this paper, we will focus on the techniques of ensuring the reliability of a web services-oriented system by selecting reliable web services components in a cost-effective manner. By 'web services-oriented system', we mean a software system that comprises web services as components. In this paper, we adopt for *web services reliability* the same definition used by IEEE for *software reliability*, which is "the probability that software will not cause the failure of a system for a specified time under specified conditions" (IEEE, 1988). We consider the reliability of an individual web service component and its interoperability with other components in a final environment. More specifically, this paper addresses the following three research questions:

*Question 1* *How can we effectively and efficiently test remote web services?*

*Question 2* *How do we test the reliability of remote web services?*

*Question 3* *How do we test the interoperability of a remote web service in a specific software environment?*

To provide an answer to these three questions, this paper proposes a mobile agents-based approach to assist a service requester in selecting appropriate web service components. Our approach provides a more cost-efficient method to help service requesters make better decisions. The remainder of this paper is organised as follows: In Section 2 we define the problem domain. In Section 3, we present our mobile agents-based approach. In Section 4, we discuss the design and implementation of intelligent mobile agents. In Section 5, we discuss experiments. In Section 6, we present merits and limitations of our approach. In Section 7, we discuss related work. In Section 8, we draw conclusions and describe future work.

## 2 Problem domain definition

First of all, we will briefly introduce the core techniques and standards of web services to provide readers with background context. A web service is a programmable web application that is universally accessible through standard internet protocols (Ferris and Farrell, 2003). web services typically adopt a provider/broker/requester architectural model (Han *et al*., 2000): service providers register web services at service brokers; service brokers publish registered services; and service requesters search for web services from the service brokers. The essential aspect of this model is the concept of dynamic invocation: web services are hosted by service providers; and service requesters dynamically invoke web services over the internet upon an on-demand basis.

The paradigm of web services originally embraces three core categories of supporting facilities (Curbera *et al*., 2003; Roy and Ramanujan, 2001):

1 communication protocols

2 service descriptions

3 service registration and discovery.

Each category has its own *ad hoc* standard: the Simple Object Access Protocol (SOAP)[1] acts as a simple and lightweight protocol for exchanging structured and typed information between web services; the Web Service Description Language (WSDL) is an eXtensible Markup Language (XML)-based description language that is used to describe the programmatic interfaces of web services (WSDL, 2004); and the Universal Description, Discovery, and Integration (UDDI) standard (UDDI, 2004) provides a mechanism to publish, register, and locate web services. These three categories provide basic descriptive means for operational infrastructure of web services.

However, depicting a narrow definition of web services that refers to an implicit definition of SOAP+WSDL+UDDI, these three categories only guide service requesters to locate one web service. A large-scale application normally requires multiple web services to collaborate. The Business Process Execution Language for Web Services (BPEL4WS) (BPEL4WS, 2003) was thus proposed for formal specification of synergistically coordinating and organising web services into business processes.

With the basic background, we can now define the problem domain more specifically by depicting a typical problem scenario, so as to facilitate our later discussions. Suppose in a project design there are four serially connected components, say $X_1$, $X_2$, $X_3$, and $X_4$, as shown in Figure 1. To simplify the question, we make the following four assumptions:

1 Circular dependency relationship is not considered, *i.e.*, the output of the component $X_4$ does not directly or indirectly affect the input of the component $X_1$.

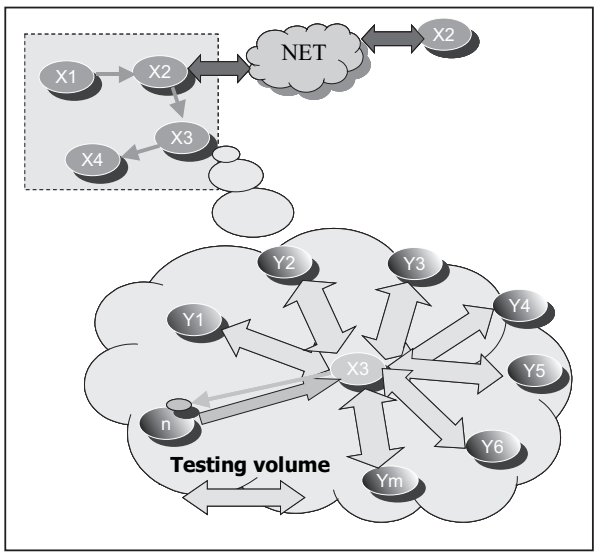
2 We assume that the requirements of each component have been predefined.

3 The subscriptions of the components represent the dependencies between the components. For example, the output of the component $X_1$ affects the input of the component $X_2$; the output of the component $X_2$ affects the input of the component $X_3$; and so on. In other words, to simplify the question, we only consider a serial dependency relationship between components without considering a parallel relationship.

4 Without losing generality, we assume that it has been decided that the component $X_2$ will be implemented by the web service $X_2$. The current task is to search for a suitable web service to fulfil the requirements of the component $X_3$. (From now on to the rest of the article, we use exchangeably the phrases 'a component' and 'a component that needs to be fulfilled by a web service'.)

After searching a UDDI public registry using the functional requirements of the component $X_3$, a set of web service candidates are found, say, $Y_1$, $Y_2$, …, and Yn. (Note that how to find a set of web services candidates based on system requirements is out of the domain of this paper.) These candidates are all published using the *ad hoc* industry standard WSDL by different services providers; thus, they may exhibit different qualities. Now the problem is how to select an appropriate web service from this list (*i.e.*, $Y_1$, $Y_2$, $Y_3$, …, Yn) to realise the functional design of $X_3$, and to ensure the overall reliability of the project. Furthermore, since this paper focuses on applying mobile agents to test web services, we will not discuss in detail how to automatically and efficiently generate test cases from WSDL.

**Figure 1** Web services-oriented scenario



## 3 Mobile agents-based approach to test web services

### *3.1 Applying mobile agents to test web services*

In order to opt for an appropriate web service, the candidate web services (*i.e.*, $Y_1$, $Y_2$, …, and Yn shown in Figure 1) should be independently tested before a decision can be made. Whether the testing stops as soon as a qualified candidate is found or until all candidates are tested is beyond the scope of this paper. What we are interested here is how to test each candidate. Since web services are remote web applications hosted by the corresponding service providers, testing has to be conducted remotely using SOAP messages through web services' interfaces published using WSDL. Nevertheless, if each test case is performed remotely over the internet, as shown in Figure 1, the testing volume can be significant thus may generate enormous network traffic. Moreover, if each test case is conducted remotely, one needs to ensure the trustworthiness of each pair of SOAP request and SOAP reply message (*e.g.*, not to be maliciously attacked in the process of internet transport) associated with each test case. This requirement is highly challenging and the possible solutions may produce significant extra overhead.

In order to eliminate both network traffic and transport protection overhead, we explore the mobile agents technology to test web services. Here we will first briefly review the basic concepts of the mobile agents paradigm, before we discuss the advantages of using the mobile agents technology in the domain of web services. Mobile agents refer to self-contained and autonomous software programs that can move from one computer to another through the network environment and act on behalf of users or other entities (Lange and Oshima, 1999; Pham and Karmouch, 1998; Rothermel and Popescu-Zeletin, 1997). Essential merits of mobile agents are: network load reduction, asynchronous and autonomous execution, dynamic adaptation, network latency overcoming, *etc*. (Lange and Oshima, 1999; Pham and Karmouch, 1998; Rothermel and Popescu-Zeletin, 1997).

Utilising the migratory characteristic of mobile agents, our strategy is to apply mobile agents to conduct web services testing. In detail, in order to test a remote web service, a mobile agent will be generated and will migrate to the remote web service candidate site carrying test cases. Then the mobile agent will conduct all the tests at the remote web service site; and the test results will be passed back. The scenario is shown in Figure 1. Suppose the number of the original SOAP request messages that need to be created to test a web service using traditional methods is *N*. When the number of test cases becomes significant, by using a mobile agent, the network traffic can be reduced from 2*N to *N* if each SOAP reply message needs to be sent back right away for evaluation, as shown in Figure 1.

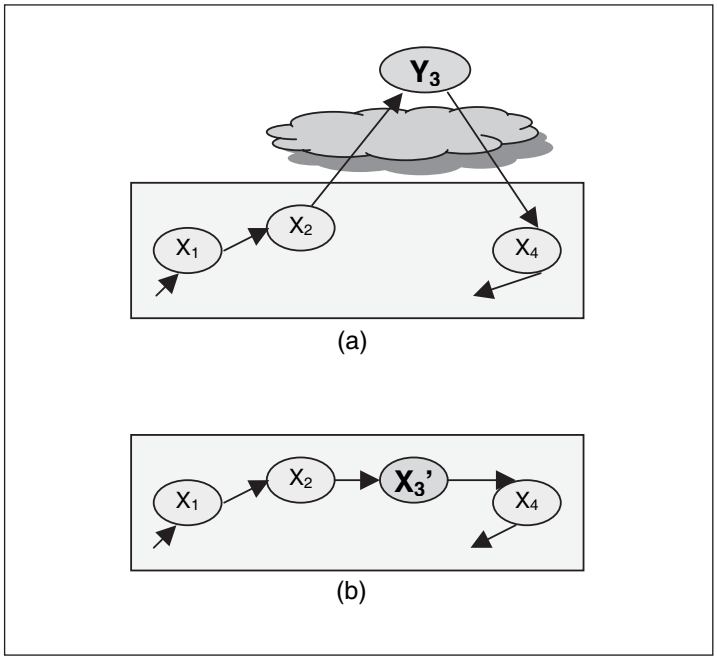
We can further enhance the efficiency of our approach by implanting intelligence into the dispatched mobile agent. Instead of merely inputting test data into the web service host under testing and sending back to the service requester test results, the mobile agent shall carry knowledge as well. In the simplest way, other than sending back each test result, the mobile agent can pack multiple results and send back in one SOAP reply message, thus further reducing network load. In more complex forms, the mobile agent will be able to calculate the reliability of the web service under testing based upon the test results and make some decision at the testing site. In other words, the mobile agents can demonstrate pro-active behaviours in the sense that they make testing decisions on behalf of their owners and based upon environmental percepts of specific web services under

testing. This means that they are able to reason and adapt their behaviours in reply to the percepts and the state of affairs. From this perspective, we say that the mobile agent is intelligent. An intelligent mobile agent can further reduce network traffic. Using the example discussed before with the number of the original SOAP request messages being *N*, the network traffic can be further reduced down to N/m, as *m* is a coefficient depending on the mobile agent's intelligence level. When *m* is large enough, our remote testing can be considered as generating insignificant network traffic.

Our mobile agents-based approach potentially benefits both web service providers and service requesters. From the web service providers' perspective, they only need to authenticate the mobile agent once when it arrives, instead of authenticating each incoming individual SOAP request message. Moreover, since the testing tasks are embedded into the mobile agents that are then dispatched to the remote web services sites and operate asynchronously at the remote sites, the service requesters can be released from being bound with every candidate web service before all the testing is finished.

Meanwhile, the system interoperability of a web service has to be fully tested before it can be adopted as a system component. In other words, the web service component being tested must cooperate with other system components in the context of a specific system environment. It is necessary to investigate how the whole system will react with every possible output from the web service component under testing. As we discussed, it is neither efficient nor practical for service requesters to be bound with service providers to examine every possible output, as shown in Figure 2(a), using the scenario that we discussed earlier where $Y_3$ is a remote web service candidate. To avoid generating significant internet traffic and facilitate the test process, we utilise mobile agents for another purpose, as shown in Figure 2(b).

**Figure 2** Interoperability testing



By sending a mobile agent to the remote web service site $Y_3$ to perform a set of tests, the test results can also be gathered by the mobile agent. If the set of test data is comprehensive enough in respect to the real operational profiles, which refer to the possible execution scenarios in the context of the final system environment, the collection of the tuples of the test data and the corresponding test results can be used to simulate a substitute component $X_3$' for the remote web service $Y_3$ in the final system, as shown in Figure 2(b). Here we use the term *substitute* because from the system's perspective, the collection of the tuples can produce exactly the same results upon the same input data, thus can functionally act exactly like the real remote web service. (Here we omit the possible performance difference due to internet transport.) In other words, we intend to employ mobile agents to carry back the full states of the remote web service candidate, so as to perform system interoperability test locally, as shown in Figure 2(b). The phrase *states of a web service* here refers to the collection of tuples of the input data and the corresponding output data of the web service over the entire input data space, which can be denoted as follows:

$$S(ws) = \bigcup_{i=1}^{n} (x_i, y_i), y_i = f(x_i), x_i \in X$$

where:

$S$ = the states of the web service named *ws*
$f$ = the functionality of the web service, which generates a unique output over an input
X = the entire input space of the web service according to its operational profile.

In summary, we propose to create mobile agents that carry test cases to remote web services sites to conduct on-site testing and facilitate web services interoperability testing. The critical challenge of this approach is how to effectively and efficiently prepare test cases to equip mobile agents.

### *3.2 Focus on eliminating web services candidates*

When a mobile agent is dispatched to a remote web service site, it will conduct test cases on behalf of the corresponding service requester. Instead of proving the correctness of a web service, our main strategy goes from the opposite direction: we focus on breaking a web service under test and eliminating it from the corresponding candidate list. In other words, our focus in this research is to eliminate web services candidates when they fail the test in terms of their reliability.

If at some point, a mobile agent at one candidate site finds that the web service being tested fails to meet a predefined minimum threshold of reliability requirement, the test does not need to continue. In other words, before all the testing is completed, if we can decide that a candidate web service is not the one we are looking for, the test may be discontinued. In yet other words, we say that the mobile agent is intelligent. Our strategy can potentially benefit both service requester and service provider. From the service requester's perspective, this method can shorten decision time and lessen the number of mobile agents that the requester needs to monitor in a certain time frame (*i.e.*, when multiple mobile agents are sent simultaneously to multiple remote web services sites). From the service provider's perspective, this strategy can alleviate traffic so that the web

service may support more simultaneous accesses (*i.e.*, the service provider may be simultaneously investigated by multiple service requesters). Although a web service may be quite heavy computationally, ideally it should be able to support multiple requesters. It should be noted that due to numerous unpredictable factors (*e.g.*, network traffic, malicious attack, *etc*.), one cannot generalise from the failure of one single test case. Hence, one single failure should not be a reason to outrightly exclude a web service candidate. Instead, an application-specific threshold is set to guide a mobile agent when a web service candidate can be safely discarded.

In order to find out if a decision can be reached before a whole set of test cases is completed, we chose to use an assertion-based technique. Software assertions are Boolean functions that can make TRUE evaluations when a program state satisfies some semantic conditions, and FALSE if otherwise (Mueller and Hoshizaki, 1994). If an assertion makes a FALSE evaluation, the execution of the program will be considered as a failure, even if the output for that execution is correct. Using the idea of assertion carefully designed segments of code are embedded into mobile agents to migrate to tested web services sites. A simple example of assertion is that the response time of a testing web service should be less than or equal to two seconds. Apparently it is neither efficient nor effective to base the reliability of a web service on a single instance of test failure. In other words, we do not wire each test case with one specific assertion. For example, the assertion example above can be refined to that a web service will be considered as timely if 90% of test cases show response time less than or equal to two seconds.

In summary, contrary to the fact that assertions have been extensively used to prove program correctness, we apply assertions to facilitate the selection process of reliable web services. In addition, we use assertion technique to ensure the aliveness of mobile agents, a topic which will be discussed next.

### *3.3 The trustworthiness of applying mobile agents to test web services*

Our mobile agents-based approach can facilitate the testing and selection of reliable web services. However, a corresponding trustworthiness issue arises and requires serious considerations.

First of all, one may argue that a web service provider might never like to have mobile agents execute locally due to security concern among other reasons. Generally, this concern is reasonable. However, there are situations where mobile agents are necessary. For example, a service requester may have to test the performance quality of a web service regardless of internet transportation. Under this condition, any remote testing cannot hamper internet communication. Therefore, the only solution is for the service requester to send a mobile agent to the web service site to conduct performance testing on site. Since such a kind of situation is unavoidable, we need to explore how to guarantee the trustworthiness of mobile agents sent to test web services.

As soon as mobile agents are dispatched to the internet, they may become untrustworthy due to attacks from malicious or errant web service hosts, or intolerable time delays caused by erratic internet behaviours or resource scarcity (Mueller and Hoshizaki, 1994). Therefore, agent owners may question the trustworthiness of the information carried back by a mobile agent. Ensuring the trustworthiness of applying mobile agents to test web services reliability is four-fold:

1 How to ensure that the incoming mobile agents do not maliciously attack the web service hosts being tested?

2 How to ensure that there are no malicious attacks to mobile agents from remote web service hosts?

3 How to ensure the trustworthiness of messages returned by mobile agents?

4 How to ensure that a dispatched mobile agent is still alive?

The first two are mainly security concerns for mobile agents against hosts with uncertain trustworthiness. These are known issues for mobile agents. A comprehensive overview of the issues and some proposed solutions are presented in Claessens *et al*. (2003). Since these two issues are not the concern of our research, we adopted a *one-time proxy signature* method for both mobile agents and agent platforms to address the two issues. This method was proposed by Kim and colleagues (Kim *et al*., 2001): a web service provider generates a proxy key pair, and signs one and only one message with the pair. A service requester, who is also the agent owner, then generates a message accordingly. This signature will be carried by the mobile agent to the web service provider's site. Both the signature of the web service provider and the signature of the service requester will be verified. Detailed discussions about the method can be found in Kim *et al*. (2001).

Regarding the third issue, when a service requester receives test results, it needs to verify whether the information comes from a mobile agent that it dispatched, and from which mobile agent that was dispatched. Therefore, each mobile agent dispatched needs to have its unique identification. Our strategy is to use a pair of unique keys for each mobile agent. When a service requester creates a mobile agent, it assigns a pair of unique session keys to the mobile agent accordingly: one public session key and one private session key. When the mobile agent is sent out, it will carry with it its public session key only. When the mobile agent prepares test results, it will encrypt the information with its public session key. When the service requester receives the returned results, it can use the unique private key of the mobile agent to decrypt the data. Since only the service requester holds the private key of the mobile agent, it will be the only one that can decrypt and understand the returned messages, and the service requester can verify if the returned messaged came from a mobile agent it dispatched.

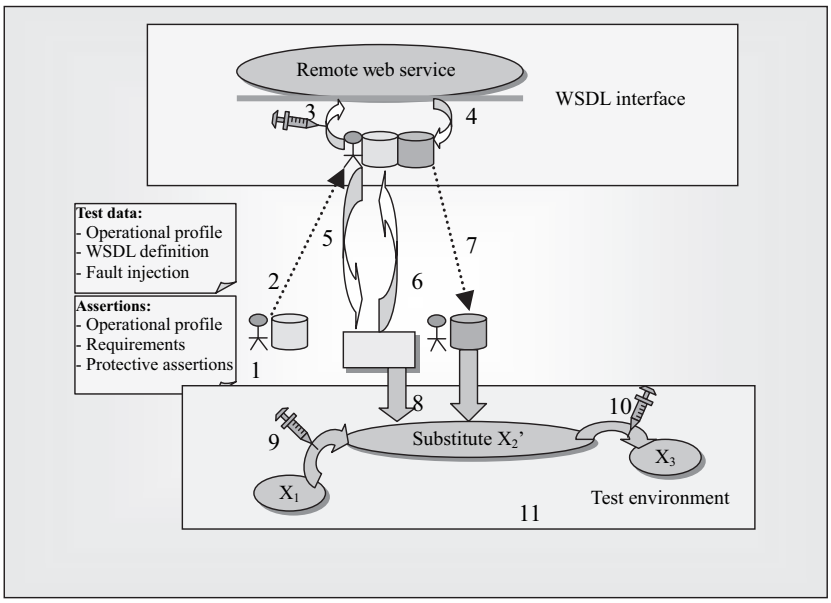
Regarding the fourth issue, after a mobile agent is dispatched, its sender (*i.e.*, the service requester) needs to verify whether it is still functioning, so as to decide the subsequent operations, *e.g.*, whether to send a new mobile agent or not. Our strategy is to enforce mobile agents to periodically send back notifications to their dispatchers to inform their senders of their condition. Then our question becomes when and how often to send notifications, taking into consideration network traffic and efficiency. We adopted a timer technique to decide the notification frequency. When a service requester prepares a mobile agent, a segment of code will be embedded to assert on a predefined timer. After a mobile agent is dispatched, its timer is triggered to assert the existence of the mobile agent and send back the notification to the service requester. The service requester then analyses the notification and decides whether the mobile agent is still functioning. If no notification comes from a dispatched mobile agent after a certain time frame, the mobile agent will be considered dead, and a new mobile agent will be created and resent. The technique of using session keys to strengthen the trustworthiness of mobile agents for the third issue can be reused here to fortify the notifications. In more detail, each notification sent back by a mobile agent is encrypted by its public session

key, and can only be decrypted by its sender using its private session key. It should be noted that the notification frequency is predefined by the service requester, *i.e.*, the sender of the mobile agent.

### *3.4 The procedure of mobile agents-based approach*

Now we are ready to present the detailed steps of how our mobile agents-based approach is used for efficient reliable web service components testing and selection. Our approach is illustrated in Figure 3 in a stepwise procedure. We will walk through the scenario step by step. The numbers marked in Figure 3 denote the order of each step in the procedure. In Figure 3, a mobile agent is represented by a pictogram, which migrates from service requester's site to the remote web service site, and then comes back carrying test results. We utilise normal public/private key technique for authorisation and authentication interactions between mobile agents and web service sites.

**Figure 3** The procedure for reliable web services selection




*Step 1*

A service requester creates an intelligent mobile agent to test a remote web service. The agent can be programmed in some mobile agent programming languages. We used Aglets (http://aglets.sourceforge.net/) since it is an open-source library and we were familiar with it from other previous research. The agent will carry several categories of data:

- test data
- assertion conditions
- unique agent public key.

As shown in Figure 3, these data are stored in an embedded database carried by the created agent. The production of the test data is based upon:

- the WSDL description of the web service
- the data perturbing technique
- the operational profile of the service requester.

A proxy, as shown as a rectangle at the service requester site, is created at the service requester site for each mobile agent created, whose functionality is to monitor the assertions sent back by the mobile agent, and decide how to act accordingly. If the mobile agent does not send information back for a predefined time period, the proxy may actively query the mobile agent to ensure that it is still alive, or simply decide to discard the mobile agent and create a new one. A specific service requester can decide to adopt either option.

*Step 2*

The service requester dispatches the intelligent mobile agent to the remote web service site. As shown in Figure 3, the mobile agent travels to the web service site together with test data and algorithms.

*Step 3*

The mobile agent generates SOAP request messages and send to the web service to test the reliability of the web service. The carried test data intend to test the reliability of the web service, and collect information on the state of the web service. As shown in Figure 3, test data are inputted to the web service.

*Step 4*

The mobile agent receives SOAP response messages sent by the web service for every test case.

*Step 5*

The mobile agent tests the assertions for the functionality and vulnerability of the web service, as well as its own trustworthiness at the moment. Assertion results are stored into a log database associated with the mobile agent, and forwarded back to the corresponding proxy at the service requester site. At this point, the mobile agent can make certain decisions based upon the assertion results.

*Step 6*

The proxy analyses the assertions returned by the mobile agent. If the proxy finds that the web service does not meet the requirements of the desired service (*i.e.*, functional requirements and vulnerability requirements), it will send a command to the mobile agent to terminate its tasks, and remove the web service from the service requester's candidate list. The process will end afterwards. If the proxy finds that the mobile agent has been maliciously attacked, it will send a command to the mobile agent to terminate its tasks, records the failure information, and goes back to Step 1 to create another mobile agent. If

the proxy finds that similar situation happens repeatedly (*e.g.*, if the number of times that the mobile agent fails for the same reason of malicious attacks surpasses some predefined threshold), it will consider the web service as untrustworthy and removes it from the service requester's candidate list.

*Step 7*

Eventually the mobile agent finishes all the test cases at the remote web service site, and migrates back to the service requester site together with log database.

*Step 8*

The service requester creates a substitute component $X_3$' for the remote web service at the local test environment based upon the log database of the returned mobile agent and the information from the associated proxy.

*Step 9*

The service requester starts system integration test to verify whether the system can tolerate the substitute component, by testing the faulty propagation of the substitute component. As shown in Figure 3, faulty data will be injected as input data to the substitute component. As a matter of fact, this fault injection has been performed at the time of designing the test data at Step 1.

*Step 10*

Faulty data will be injected as output data of the substitute component to its subsequent component $X_4$.

*Step 11*

The service requester monitors the output data from the component $X_3$' and subsequent components, to verify whether the errant action of the substitute component will be propagated to affect the whole system.

## 4 Design

In short, a intelligent mobile agent can be dispatched to test a remote web service as a stand-alone application, and carry back test logs (*i.e.*, test data sets and the corresponding test result sets) of the web service to form a local substitute component to further test the interoperability of the component in the final system environment. In this section, we will discuss the internal structural design of an intelligent mobile agent.

Figure 4 illustrates the internal structure of a mobile agent. Five functional components are identified:

1 a control manager

2 a test manager

3 an assertion manager

4 a SOAP message generator

5 a SOAP message interpreter.

Three databases are identified:

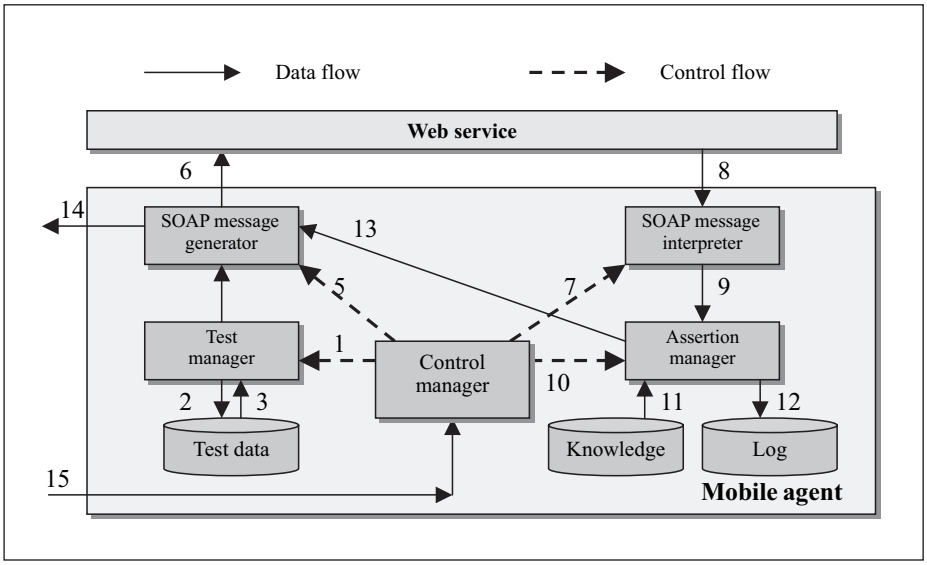
1 a test database

2 a knowledge database

3 a log database.

The control manager controls all other modules, *e.g.*, decides when to start testing, when to send back assertions, when to terminate execution, *etc*. The test manager loads test data from the test database, and passes them to the SOAP message generator. The SOAP message generator component generates SOAP request messages and sends to the web service. The SOAP message interpreter module listens to the SOAP response messages from the web service, interprets the responses to obtain test results, and passes the test results to the assertion generator. The assertion generator pairs the test data and corresponding test results together, utilises the knowledge database to make assertion tests, and writes to the log database. In other words, the assertion manager is the reasoning component of the mobile agent. If a certain assertion is false, the assertion information may be passed to the SOAP message generator to create a SOAP message and will be sent back to the service requester.

**Figure 4** Internal structure of an intelligent mobile agent




The test database may include predefined discrete test data set. The knowledge database provides information to help the assertion manager make decisions based upon whether some predefined semantic assertions are satisfied or not. The log database will store the pairs of test data and corresponding test results, as well as assertion information.

Figure 4 also exhibits the scenarios of how a mobile agent works. The arrowed lines represent the flow of information between the components: the solid directed lines indicate data flows; and the dotted directed lines indicate control flows. The numbers associated with each line represent the order of the scenario. We will walk through the scenario as shown in Figure 4.

*Step 1*

The control manager starts the mobile agent by sending control information to the test manager.

*Step 2*

The test manager queries the test database to iterate through the predefined test data set.

*Step 3*

The test database retrieves information from the test manager.

*Step 4*

The test manager generates test cases and passes them to the SOAP message generator.

*Step 5*

The control manager synchronises the SOAP message generator to work.

*Step 6*

The SOAP message generator creates SOAP messages and sends them to the web service.

*Step 7*

The control manager starts the SOAP message interpreter to listen to the responses from the web service.

*Step 8*

The SOAP message interpreter catches reply information from the web service.

*Step 9*

The SOAP message interpreter analyses incoming SOAP response messages, obtains test result data, and passes them to the assertion manager.

*Step 10*

The control manager initiates the assertion manager to work.

*Step 11*

The assertion manager retrieves information from the knowledge database to make decisions on assertions.

*Step 12*

The assertion manager stores in the log database pairs of test data and their corresponding test results, as well as the assertion results.

*Step 13*

If an assertion results in a false, the assertion manager will send the information to the SOAP message generator.

*Step 14*

The SOAP message generator generates a SOAP message and sends back to the service requester.

*Step 15*

Based upon the assertions received, the service request may send a message to the control manager to terminate the mobile agent's task. This message may command the mobile agent to travel back to the service requester with information gathered so far, or request the mobile agent to terminate itself (*e.g.*, if the service requester suspects that the mobile agent has been maliciously attacked so that the information it carries is useless).

## 5 Experiments

In order to examine the effectiveness of our proposed mobile agents-based testing and selection approach, we carried out a series of simulations to study the system performance. An environment was set up to simulate the scenario described in Figure 1. The final system intends to be a student registration and record system, where students can register and retrieve course grades online. The system was designed to be composed of a sequence of components, each being implemented by a web service. The component $X_2$ accepts a student's social security number (string) and outputs her unique student ID (string). A web service was constructed to realise this function. The component $X_3$ intends to receive a set of input arguments – a student ID (string), course ID (string), term (string), and year (integer) – and checks databases to return the corresponding course credit hours (integer) and the grade (double). The data retrieved from the database will be in the format of data type double. The service will then translate it into a character in the range of A~F. Three web service candidates $Y_1$, $Y_2$, and $Y_3$ implemented the same WSDL definition.

Our previous research on web development yielded a student record module (Zhang and Chung, 2003) built upon Java 2 Platform Enterprise Edition (J2EE)[2] technology. In order to build a web service candidate, we reused the module by wrapping it with a web service interface. To build three web service candidates, we:

1 obtained three copies of the module

2 deployed the three copies to three machines

3 wrapped each one with a web service interface.

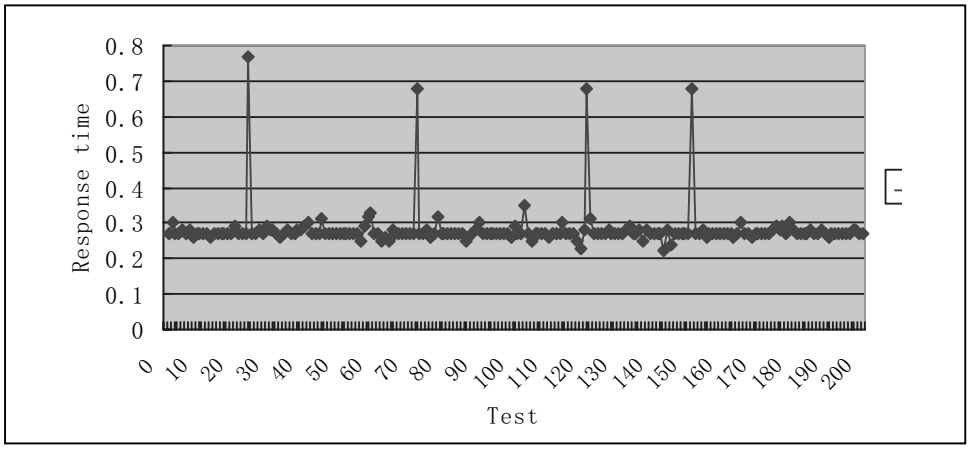
The three candidates were deployed to different machines with different computational power, thus the performance of each candidate was different. The first machine (M1) was a 0.8GHz/250MB Intel Pentium III machine running Microsoft Windows 2000; the second machine (M2) was a 1.4GHz/500MB Intel Pentium III machine running Microsoft Windows XP; and the third machine (M3) was a 1.8GHz/1GB Intel Pentium III machine running Microsoft Windows XP. The machine that acted as a service requestor (MC) was a 1.8 GHz/1GB Intel Pentium III machine running Microsoft Windows XP.

For performance comparison, we designed the following test sets:

*Experiment 1 Test the response time of a web service without mobile agents.*

First of all, we designed experiments to test the response time of a web service candidate without using mobile agents. The machine acting as a service requestor (*i.e.*, MC) sent a SOAP request to a service candidate (*e.g.*, M3). The response time was counted from the time when the request was sent until the time when the response was received. To simplify the scenario, the time when a test case was generated (*i.e.*, a SOAP request message) and the time when a test result was analysed (*i.e.*, a SOAP response message) were not considered. With the environment set up, Figure 5 illustrates 200 test results of response time, using MC as the service requestor and M3 as the service candidate. As shown in Figure 5, the average response time was 0.28 second. To simulate the unpredictable internet traffic, we inserted random time delay of 0.01 second to 0.3 second in the web service on M3. As shown in Figure 5, a typical response time ranges from 0.22 second to 0.27 second. With random time delay, some tested response time went up from 0.33 second to 0.36 second. In addition, to simulate unacceptable time delay that might be caused by either the web service or internet traffic, we set the upper threshold time limit to 0.4 second. In other words, if a result was not received in 0.4 second, it was considered as unacceptable or lost, thus another request would be sent. As shown in Figure 5, four cases with response time around 0.7 second exhibited the situation, where the response time was calculated by adding the 0.4 second to the second result.

**Figure 5** Response time without mobile agents



To sum up, the response time (*ResponseTime(ws)*) of a test case of a web service (*ws*) is the sum of request travel time (*travel(req)*), processing time (*processing*), and response travel time (*travel(res)*). If the sum exceeds the predefined upper threshold (*threshold*), the response time would be the sum of the successful response time and the unsuccessful time (*unsuccessful*) times the upper threshold time. Formula (1) summarises the definition:

$$ResponseTime(ws) = \begin{cases} travel(req) + processing\ travel\ (res), \\ \qquad\qquad ResponseTime(ws) < threshold \\ travel(req) + processing + travel(res) + unsuccessful * threshold, \\ \qquad\qquad ResponseTime(ws) >= threshold \end{cases} \quad (1)$$

*Experiment 2 Test the efficiency of using mobile agents.*

Second, we designed experiments to prove the efficiency of using mobile agents for testing web services. To simplify the scenario, we made the following assumptions:

- We ignored the time spent in analysing returned mobile agents.
- We did not consider possible attacks to mobile agents.

Using the same environment in Experiment 1 (*i.e.*, with MC as the service requestor and M3 as the service candidate), we measured the response time of mobile agents dispatched. The response time of a mobile agent *A* (*ResponseTime(A)*) is composed of the following elements as shown in Formula (2):

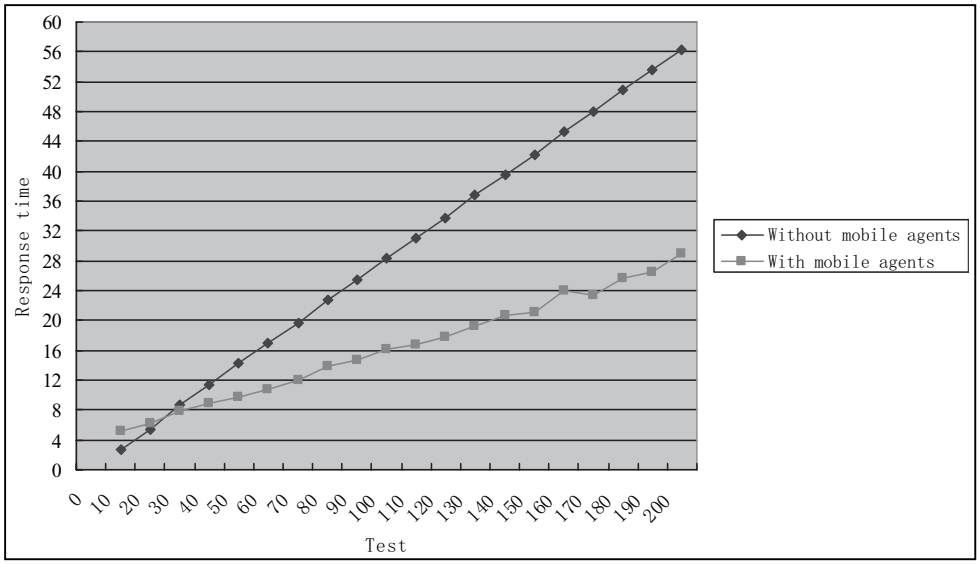
- the creation time of the mobile agent (*creation(A)*)
- the travel time of the mobile agent to the web service (*travelto(A)*)
- the processing time of the mobile agent at the web service site (*processing(A)*)
- the creation time of the message returned by the mobile agent (*creationreturn(A)*)
- the travel time of the mobile agent to the service requestor (*travelfrom(A)*):

$$\begin{aligned} ResponseTime(A) = {} & creation(A) + travelto(A) + processing(A) + \ldots \\ & + creationreturn(A) + travelfrom(A) \end{aligned} \quad (2)$$

As discussed in the previous sections, a mobile agent can be used to test a set of test cases in one dispatch. Each set of test cases includes some number of test cases, say 10, 20, 30, *etc*. In order to compare the efficiency of using and not using mobile agents, we performed the same set of test cases upon both approaches and compared their total response time. For example, if a set of test cases contained 50 test cases, a mobile agent would be created to carry the 50 test cases altogether. The response time spent by using mobile agent would then be calculated using Formula (2) as discussed above. To calculate the total response time without the use of mobile agents, 50 test cases were independently conducted, each individual response time calculated using Formula (1), and then all 50 response time were summed up to obtain the final result.

Various sizes of test cases set were created to test the two approaches. The size of the test case set ranged from as small as containing ten test cases, to as large as containing 200 test cases. We chose to adopt an iteration range of ten test cases as the size grew. Figure 6 illustrates our test results using mobile agents and without mobile agents. The total test time using mobile agents and without mobile agents are generally exhibited as two linear functions of the number of test cases conducted.

**Figure 6** Comparison of response time with and without mobile agents



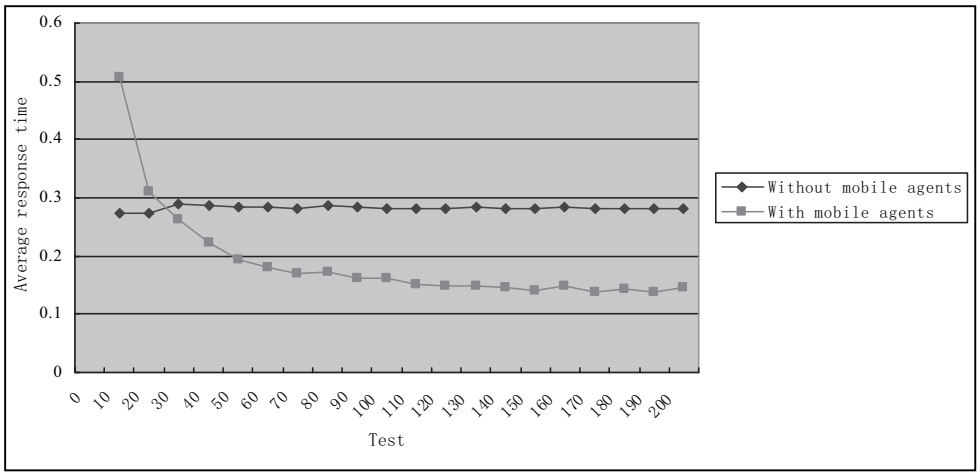

From Figure 6, we can see that when the number of test cases is small, say ten or 20, the total test time using mobile agents is greater than without the use of mobile agents. The reasons are the overhead introduced when mobile agents are used: the creation time of a mobile agent adds some fixed overhead, and the creation time of messages returned by a mobile agent also adds extra overhead. When the number of test cases is small, the internet travel time of each request and response saved using mobile agents is counteracted by the overhead introduced.

When the number of test cases increases, say above 20, as Figure 6 shows, the total test time spent using mobile agents is less than without the use of mobile agents. As the size of the set of test cases increases, the efficiency of using mobile agents becomes more pronounced. The essential reason is that mobile agents carry all test cases to the web service candidate site and all the test cases are performed at the remote candidate site; thus, all the travel time of the request message and response message of each test case is saved.

We then calculated the average response time for both approaches and summarised the results in Figure 7. For the approach without mobile agents, the average response time stabilised to be around 0.28 second. For the approach with mobile agents, as the size of the test case set increased, the average response time dropped significantly from the original 0.51 second to 0.14 second when the size of the test case set became 200. With the approach using mobile agents, Figure 7 shows that the average response time very quickly converges to be around 0.14 second when the size of the test case set reached 90,

and does not drop much at all afterwards. The reason is that for every test case, the approach using mobile agent needs to spend some overhead on preparing the test case and preparing a return message. From Figure 7, we can see that when the number of test cases is lower than 30, the overhead is significant. However, when the number goes beyond 30, the use of mobile agents exhibits significant efficiency. It should also be noted that the response time is observed as waiting time from the perspective of service requestors. Mobile agents perform the testing tasks while the service requestors are released to conduct other tasks at the same time.

**Figure 7** Comparison of average response time with/without mobile agents




We conducted the same experiment using different service candidate machines (*i.e.*, M1 and M2). The values of response time were different, due to the different machine power. However, there was an indication that the efficiency of using mobile agent remained the same even as the size of the test case set increased.

## 6 Discussion

In this section we will analyse and discuss our mobile agents-based approach, including both its merits and its limitations. Then we will envision some techniques and standards that are in demand in this field.

### *6.1 Merits*

Here we evaluate whether our mobile agents-based approach achieved the goal stated at the beginning of this paper: to provide one possible solution in selecting reliable web services components. We examine whether our method solves the four research questions.

*Question 1 How can we effectively and efficiently test remote web services?*

By applying mobile agents technology to help web services components selection, our mobile agents-based approach is more efficient by reducing the overhead incurred by invoking remote web services. Service requesters can be released from being forced to remain the bindings with every candidate web service. In addition, by dispatching multiple mobile agents to multiple candidate web services simultaneously, the parallelism can be largely increased. Furthermore, by mobile agents performing testing at the web services sites, we largely decrease the possibility of malicious attacks to every remote request. Therefore, utilising mobile agents technology can make the selection process faster, safer, and more resource efficient.

In addition, by using assertions to test the vulnerability and functional quality of remote web services, our mobile agents-based approach provides a dynamic strategy for service requesters to quickly identify the reliability of web services. This strategy can also benefit the service providers by alleviating traffic.

*Question 2 How do we test the reliability of remote web services?*

By (1) constructing test data including normal data and corrupted data, and (2) setting up assertions according to the operational profiles of service requester and the functional requirements of the desired component, our mobile agents-based approach is capable of certifying whether the tested web service can thoroughly fulfil the functional requirements as desired. In addition, by perturbing the test data to imitate unusual events, our approach is capable of testing whether the hosts of web services are acting maliciously or errantly at invocation times.

Based upon remote web services' published descriptions and service requester's operational profiles, currently our mobile agents can test the functionalities and reliability of the web services. However, our method has the potential to test more non-functional attributes of web services. For example, mobile agents can carry a matrix of test data to a remote web service to test more 'ilities'.

*Question 3 How do we test the interoperability of a remote web service in a specific software environment?*

By perturbing the output data of a web service gathered by mobile agents in a system integration test, our mobile agents-based approach is capable of testing whether a web service's errant action will affect the quality of a composed environment.

In addition, it should be noted that, although our method concentrates on testing the reliability of web services as components in an application system, this method can be used to test the reliability of stand-alone web services. That being the case, the interoperability testing can be omitted.

Furthermore, although our current research focuses on how to assist service requesters in selecting reliable web service components, this research can be potentially applied to facilitate web services certification processes.

In summary, our mobile agents-based approach provides a cost-efficient method to reveal the testability of remote web service components. Our approach also assists service requesters in making better decisions.

### *6.2 Limitations*

It should be noted that for our mobile agents-based approach to be active, there is a need for web service providers' sites to authenticate incoming mobile agents. The web services' sites have to trust the mobile agents before letting them perform tests locally. There is a potential risk that our mobile agents-based technique might be abused in a distributed denial-of-service attack. There could be ways to limit the danger up front, such as authentication. Since mobile agents-based testing can greatly facilitate web services testing as discussed in the paper, we believe that this approach has a great potential. Security issue is currently the biggest obstacle. In addition, to reduce risks on both sides of service provider and requester, there clearly needs to be some coordination on the kinds of agents that are acceptable at a remote site and how those agents must behave. Our current research addresses some solutions to this issue and we will explore more in our future work.

At this stage, we suggest a stage-by-stage acceptance of our approach. The first choice is the acceptance of generalised test results and agents. The second choice is a 'registration' of agents with custom tests and posting of test results after a single canonical run. The third choice is a real-time testing via agents. All of these choices would have to be coordinated with the site-trust issues as we discussed earlier.

In order to make our mobile agents-based approach more effective and feasible, we need several code generation tools. First, the mobile agent code needs to be automatically generated, or partially automatically generated. Second, we need a test data generator that can not only generate normal test data according to expected operational profiles, but also create corrupted test data that imitate anomalous events, including malicious attacks.

Our mobile agents-based approach might not apply to some types of web services. For example, considering a web service such as flight tickets reservation, it is impractical to use mobile agents to test its quality due to associated charges. Under this kind of circumstances, the reputation of the specific web service provider may be the main criterion for the decision.

### *6.3 Further demand*

In the future, more powerful web service specification languages should allow service providers to publish their test suites with test data and corresponding results. In other words, the whole set or partial set of the log (*i.e.*, the state of the remote web service) that our mobile agents try to obtain can be acquired from a published site. That being the case, the mobile agents will merely need to experiment on the test data that are not found in the published test cases. Therefore, significant amount of efforts can be saved. Of course, the prerequisite of this approach is that the test cases published by the service providers should be trustworthy. The reputation of the service providers or some certification agencies can help on this issue.

Current publication and description languages for web services (*e.g.*, WSDL) do not provide the facilities to define the nonfunctional features of web services, such as Quality of Service (QoS) features, constraints, web service provider's reputation, *etc*. We envision that extensions will be added to the web services publication and description languages.

## 7 Related work

OASIS' WS-Reliability (2004) provides a standard to ensure reliable message interactions between web services. Contrary to this standard that targets reliable messaging at the transport level, our research targets the analysis and testing of reliability of a web service itself.

Casati *et al*. (2004) suggested that web service providers define *service quality matrix*, which contains nonfunctional parameters specifying the cost, duration, and other characteristics of a service to help service requestors make decisions over multiple candidates. However, their work remains as a high-level abstraction without technical discussions such as how to construct a *service quality matrix*.

Maximilien and Singh (2004) proposed to adopt dedicated agents to gather, store, aggregate, and share QoS data in order to help dynamic service selection. A layered QoS ontology is presented: the upper ontology defines basic QoS concepts such as quality, its attributes, measurement, *etc*.; the lower layer defines QoS aspects such as availability, capability, interoperability, *etc*. In contrast with their work, we use agents for different purposes. Their agents serve as independent information centres that provide QoS data for service requestors; our agents are dynamically created by service requesters and travel to service candidates to gather QoS information at run time.

Tsai *et al*. (2004) believe that all web services parties – including service providers, service brokers, and service requestors – must collaborate to perform web services testing. The essential technique of their collaborative group testing approach is to construct trustworthy service brokers and utilise distributed agents to rank and vote for appropriate services based upon service histories. Tsai *et al*. proposed to equip UDDI servers with check-in and check-out verification facility so that web services are validated and verified before they are published on the UDDI servers. In contrast with their work, our approach intends to test already-published web services.

Voas and colleagues introduced an advanced fault injection technique called Interface Propagation Analysis (IPA) to test upon blackbox-like software systems (Voas and McGraw, 1998). IPA technique injects corrupted data to the input of a blackbox system, and monitors the output of the system to obtain knowledge of its fault tolerance. In this research, we apply the basic concept of IPA to test both the fault tolerance of web services as stand-alone web applications and the system integration interoperability of web services serving as components in larger systems. However, the IPA technique is rather a high-level guideline than a concrete methodology. In the context of web services, we investigate how to utilise corrupted data to eliminate web services candidates that are not qualified and to test the interoperability of web service components.

Kassab and Voas (1998) proposed to adopt the fault injection technique to fortify mobile agents. Faults are injected into mobile agents to obtain higher observability. Compared to their work, we adopted the fault injection technique for different purposes. Contrary to their work that uses the fault injection technique to safeguard mobile agents, we equipped mobile agents with fault injection stingers to poke at remote web service sites to:

- test the fault tolerance of remote web services
- obtain full states of remote web services so as to facilitate the local interoperability testing with remote web services as components

In addition, the faulty data they injected into mobile agents intend to fortify mobile agents themselves; thus, they were generated based upon the internal code of mobile agents. On the other hand, our faulty data injected into mobile agents intend to test remote web services; thus, they are generated based upon the operational profiles of the local system and the interfaces of the remote web services published in WSDL.

## 8 Conclusion and future work

In this paper we propose a mobile agent-based, fault-injection-equipped, and assertion-oriented approach to help effectively and efficiently select appropriate web service components to ensure the reliability of a software system. Our future work will include:

- constructing code generation tools for mobile agents and their test data, including both normal test data and corrupted data
- exploring more selection criteria to ensure more nonfunctional attributes of web services
- conducting more case studies.

## Acknowledgements

The author would like to thank the guest editors Professor Patrick C.K. Hung and Dr. Casey K. Fung for organising this special issue. The author also deeply appreciates Dr. Jeff Voas for the initial inspirational discussion that directly led to this research.

## References

Business Process Execution Language for Web Services (BPEL4WS) (2003) *Specification: Business Process Execution Language for Web Services Version 1.1*, 5 May, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

Casati, F., Castellanos, M., Dayal, U. and Shan, M-C. (2004) 'Probabilistic, context-sensitive, and goal-oriented service selection', *Proceedings of the 2nd ACM International Conference on Service Oriented Computing*, New York, NY, USA, pp.316–321.

Claessens, J., Preneel, B. and Vandewalle, J. (2003) 'How can mobile agents do secure electronic transactions on untrusted hosts? A survey of the security issues and the current solutions', *ACM Transactions on Internet Technology (TOIT)*, February, Vol. 3, No. 1, pp.28–48.

Curbera, F., Khalaf, R., Mukhi, N., Tai, S. and Weerawarana, S. (2003) 'The next step in web services', *Communications of the ACM*, October, Vol. 46, No. 10, pp.29–34.

Ferris, C. and Farrell, J. (2003) 'What are web services?', *Communications of the ACM*, June, Vol. 46, No. 6, p.31.

Fremantle, P., Weerawarana, S. and Khalaf, R. (2002) 'Enterprise services', *Communications of the ACM*, October, Vol. 45, No. 10, pp.77–82.

Gold, N., Knight, C., Mohan, A. and Munro, M. (2004) 'Understanding service-oriented software', *IEEE Software*, March–April, pp.71–77.

Han, R., Perret, V. and Naghshineh, M. (2000) 'WebSplitter: a unified XML framework for multi-device collaborative web browsing', *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW)*, Philadelphia, PA, USA, pp.221–230.

Holland, P. (2002) 'Building web services from existing application', *eAI Journal*, September, pp.45–47.

Institute of Electrical and Electronics Engineers (IEEE) (1988) 'IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software', *IEEE Std 982.1-1988*, Institute of Electrical and Electronics Engineers.

Kassab, L. and Voas, J. (1998) 'Agent trustworthiness', *Proceedings of 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations*, Springer, 20–24 July, pp.121–133.

Kim, H., Baek, J., Lee, B. and Kim, K. (2001) 'Secret computation with secrets for mobile agent using one-time proxy signature', *Proceedings of the 2001 Symposium on Cryptography and Information Security*, pp.845–850.

Lange, D.B. and Oshima, M. (1999) 'Seven good reasons for mobile agents', *Communications of the ACM*, Vol. 42, No. 3, pp.88–89.

Maximilien, E.M. and Singh, M.P. (2004) 'A framework and ontology for dynamic web services selection', *IEEE Internet Computing*, Vol. 8, No. 5, pp.84–93.

Mueller, B.A. and Hoshizaki, D.O. (1994) 'Using semantic assertion technology to test application software', *Proceedings of Quality Week*, May.

Neumann, P. (2004) *Principled Assuredly Trustworthy Composable Architectures, Emerging Draft of the Final Report for DARPA's Composable High-Assurance Trustworthy Systems (CHATS) Program*, http://www.csl.sri.com/users/neumann/chats4.pdf

Parnas, D.L., Schouwen, A.J.V. and Kwan, S.P. (1990) 'Evaluation of safety-critical software', *Communications of the ACM*, June, Vol. 33, No. 6, pp.636–648.

Pezzini, M. (2003) *Composite Applications Head Toward the Mainstream*, 16 October, http://www.gartner.com

Pham, A. and Karmouch, A. (1998) 'Mobile software agents: an overview', *IEEE Communications Magazine*, July, Vol. 36, No. 7, pp.26–37.

Rothermel, K. and Popescu-Zeletin, R. (Eds.) (1997) 'Mobile agents', *Lecture Notes in Computer Science Series*, Vol. 1219.

Roy, J. and Ramanujan, A. (2001) 'Understanding web services', *IEEE IT Professional*, November, pp.69–73.

Stal, M. (2002) 'Web services: beyond component-based computing', *Communications of the ACM*, October, Vol. 45, No. 10, pp.71–76.

TechMetrix/SQLI (2003) *Adoption of Web Services and Technology Choices*, February, http://www.techmetrix.com/products/publiDetail.php?code=rep0203-1

Tsai, W.T., Chen, Y., Paul, R., Liao, N. and Huang, H. (2004) 'Cooperative and group testing in verification of dynamic composite web services', *Proceedings of 28th Annual International Computer Software and Applications Conference – Workshops and Fast Abstracts – (COMPSAC)*, September, pp.170–173.

Universal Description, Discovery, and Integration (UDDI) (2004) *Universal Description, Discovery, and Integration*, http://www.uddi.org/

Voas, J. and McGraw, G. (1998) *Software Fault Injection: Inoculating Programs Against Errors*, New York: John Wiley and Sons, ISBN 0–471–18381–4.

Web Services Description Language (WSDL) (2004) *Web Services Description Language*, http://www.w3.org/TR/wsdl

WS-Reliability (2004) http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm

Zhang, J. and Chung, J-Y. (2003) 'Mockup-driven fast-prototyping methodology for web application development', *Software Practice and Experience Journal*, Vol. 33, No. 13, pp.1251–1272.

## Notes

1 SOAP 'Simple Object Access Protocol (SOAP) 1.1.'

2 Java 2 Platform Enterprise Edition (J2EE), http://java.sun.com/j2ee.