Verification of Web Services Using an Enhanced UDDI Server

W. T. Tsai, R. Paul*, Z. Cao, L. Yu, A. Saimi, B. Xiao
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287

*Department of Defense
Washington, DC

Abstract

The UDDI (Universal Description Discovery and Integration) provides classification to find the distributed Web Services (WS) by keyword matching. The UDDI version 3 allows searching WS using digital signatures. However, it still needs systematic verification to ensure WS quality in a timely fashion. This paper proposes adding verification mechanism to the UDDI servers including check-in and checkout of WS. The key idea is that test scripts should be attached to WS, and these test scripts will be used by both WS providers and clients. Before accepting a new WS into the service directory, the new WS must be tested by the associated test scripts, and they will be accepted only if the test was successful. Before using a specific WS, a client can use the appropriate test scripts to test the WS and it will be used only if the test was successfully. While the code for WS may be not available, but the associated test scripts can be openly available. This paper also suggests test script specification techniques and distributed test execution techniques to perform testing with a UDDI server.

Keywords: Web services, UDDI, verification, scenario-based testing, and distributed test execution.

1. Introduction

Web Services (WS) provide a new architecture/paradigm for building distributed computing applications based on XML. It provides a uniform and widely accessible interface to glue the services implemented by the other middleware platform over the Internet by utilizing standard Internet protocols such as WSDL [19], SOAP and UDDI [8]. Figure 1 shows the general WS architecture [8].

1. The WS provider registers its service with a registry that is maintained by the service broker. The service broker represents a set of software interfaces (a registry service) for published WS.
2. The requester makes a call to the broker's UDDI registry, seeking a desired service and instructions on how to use the service.
3. Once the requester finds the right service, the service broker returns the service's details to the requester.
4. The requester invokes the service by making an SOAP call to the service provider.
5. Finally, the provider delivers application results to the requester. The transaction is now completed.

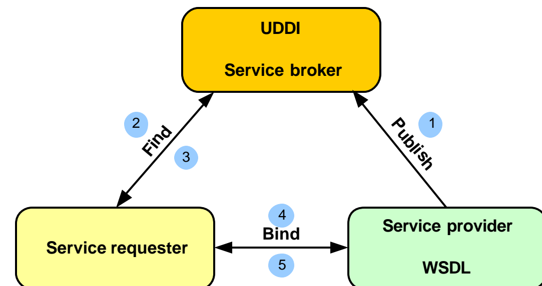

Figure 1. General Web Services Architecture

One important issue is to make WS highly reliable and robust. Testing may be one of most practical and useful quality assurance techniques [2][9][11][12]. However, testing WS is difficult due to the following reasons:

?? WS have a loosely coupled architecture but still demands high assurance;
?? WS can be invoked by unknown parties with unpredictable requests by other WS or clients;
?? WS have a dynamic runtime behavior because they involve discovery and binding with multi-parties at runtime including middleware and other WS.
?? Computing WS may involve concurrent threads and object sharing, and testing concurrent processes is difficult.

Standard Internet protocols such as WSDL and UDDI were designed mainly for their functional features with only minor consideration in verification. For example,

UDDI version 3 started addressing this issue by incorporating digital signatures and test environment [18]. Version 3 proposes public, private and shared repositories to limit the WS usage, i.e., using private repository to protect the development versions. Also, to prevent malicious or false services, UDDI specification adopts the digital signature techniques by W3C to guarantee the WS's quality. In this way, only those WS that have signed digital signatures will be used. To facilitate WS testing, WSDL can be extended to include input-output dependency, invocation sequence, hierarchical functional description and concurrent sequence specifications [13]. However, many issues still remain open.

First, the UDDI server may need to ensure that WS registered at its place have certain quality in addition to digital signatures provided by UDDI version 3. This is between the UDDI server and WS providers.

On the other hand, the UDDI may need to indicate to its clients that WS registered must have certain quality in addition to knowing which provider actually delivered the software. This is between the UDDI server and WS clients.

Both of the above two issues can be addressed by attaching test scripts to the UDDI server. Each WS must have its own test scripts, and when registering with the UDDI server, the WS providers need to provide both the software and associated test scripts. While the WS source code may not be open to clients, its test script source code can be made available to clients to increase their confidence in using the WS. In this way, a WS client can use the associated test scripts to test the WS before using the WS.

Furthermore, the UDDI server can maintain a list of hierarchical domains and sub-domains, and attach associated test scripts to each domain or sub-domain. The domains or sub-domains will be hierarchically arranged similar to an OO inheritance tree, with a sub-domain satisfing all the properties of its parent domains. When a service provider wishes to register its WS to the UDDI server, it needs to indicate which sub-domains that the WS will provide, and the WS must pass the entire test scripts associated with the sub-domain, the parent domain of the sub-domain, and their parent domains in this manner all the way to the root. In this manner, WS registered at the UDDI server will have certain minimum assurance for both service providers and clients.

This paper proposes the following features to enhance the verification of WS with UDDI:

?? The UDDI server will store test scripts in addition to WS, and test scripts will be arranged in a hierarchical manner parallel with a hierarchical tree of domains and sub-domains;

?? The UDDI server will perform a *check-in* mechanism to accept and store WS from a service provider (Figure 2). Essentially, the UDDI server will accept a service if it passes all the test scripts associated with identified sub-domains and their parent domains.

?? Similarly, the UDDI server will also perform a *checkout* mechanism to release WS to a service client (Figure 2). A client can use the associated test scripts to test-drive the WS before using them in applications. Figure 2 shows the check-in and checkout parties.

?? The test scripts associated with a domain will follow typical scenarios in that domain. Section 3 will present test script generation.

?? The UDDI server will have a test infrastructure consisting of test masters, agents, and monitors to perform WS testing remotely. This is discussed in Section 4.

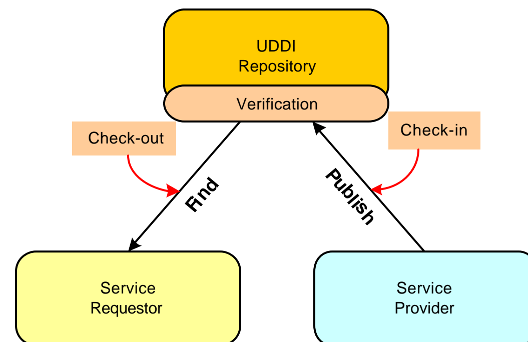

Figure 2. Check-in and Checkout of Web Services

This paper is organized in the following way: Section 2 presents the check-in and checkout in detail; Section 3 shows various testing scripts needed for testing WS; Section 4 presents a test infrastructure to perform distributed test execution using agents; Section 5 proposes using free computer resources on the web to perform verification of WS; and finally, Section 6 concludes this paper.

2. Check-in and Checkout of Web Services

The current UDDI specification provides well-formed description method and the API set for publisher and inquiry to operate the UDDI service. UDDI provides three types: white, yellow and green. White pages include contact information; yellow pages describe businesses and services according to taxonomies; and green pages provide the detailed technical information about WS. All data are organized in four data structures represented in XML: *BusinessEntity*, *BusinessService*, *BindingTemplate* and the *tMode* [17].

Together with a test infrastructure, the UDDI server checks-in WS in the following manner (Figure 3):

?? A WS provider (or publisher) develops WS and registers them to a UDDI registry, by creating *BusinessKey* and *BusinessEntity* objects, URL, and *CategoryBag*, and *CategoryBag* contains domain information.

?? The UDDI registry accepts the service provider's request, initiates registration process by testing the WS. The testing is performed by a test master and test agents (See Section 4).

?? The test master finds the corresponding test scripts stored in the database, sends test commands to test agents, and finally reports testing results to the UDDI registry after receiving the results from agents.

?? If the test results are positive, the UDDI registry checks-in the WS into its repository; otherwise the WS will be refused.

?? Finally, the UDDI server returns the registration results to the WS provider.

The test scripts, like the WSDL files, are often public information and a WS provider can obtain a copy of related test scripts before submitting WS for check-in.

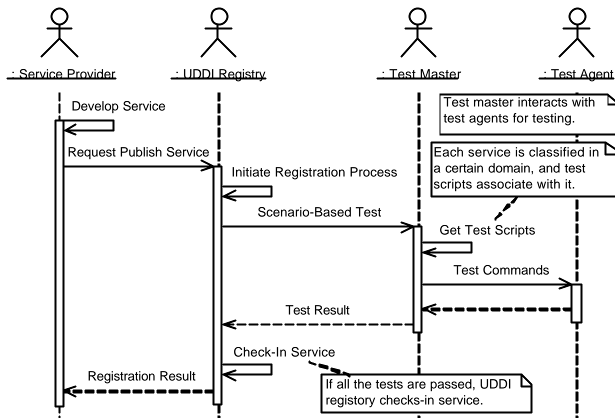

Figure 3. Check-in Sequence Diagram

Similarly, the UDDI server can checkout WS in the following way (Figure 4):

?? A WS client requests a UDDI registry about the service it needs;

?? The UDDI server will match the needs of the clients with the WS in its database. Once a set of candidate WS is identified, the UDDI server will return the candidate WS and their test scripts to the client. The candidate WS are released to the clients marked tentative because at this time the client has not experimented the WS yet;

?? The client can now experiment the candidate WS by using the supplied test scripts.

?? After experimentation, if the client is satisfied with the results, it will inform the UDDI server about the selection; and

?? The UDDI then checkouts the WS selected to the client.

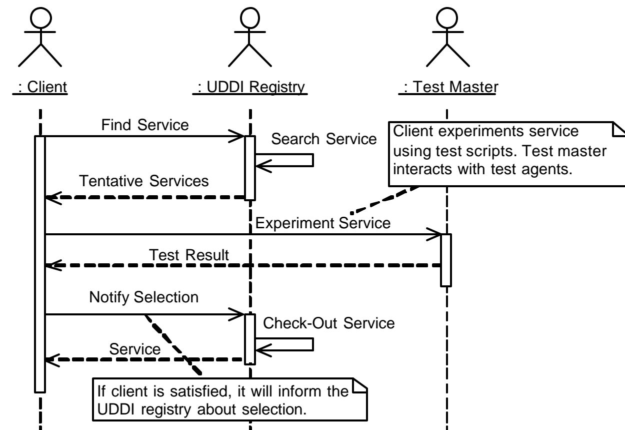

Figure 4. Check-out Sequence Diagram

One interesting aspect of the Checkout process is that it will release the WS to a client together with test scripts before the WS are officially released by the checkout process. This is needed because a client needs to run the test scripts using the supplied WS. In case the supplied WS are freely available, this is acceptable. However, in case the use of the WS requires a fee, the WS provider may not be willing to supply the WS to a client just for test drive. Several ways are possible to address this problem:

Usage count: The WS supplied during the checkout process has a usage counter, just large enough for a client to test drive the WS but not enough the WS for applications. In this way, the WS or the test infrastructure should have a counter, and the counter is decremented each time the WS is accessed. When the counter reaches zero, the WS will be no longer available.

Time count: Instead of usage count, the WS can be provided to a client for a limited time. This is patterned after common business where they allow customers to use a service for free for a limited time. When the time expires, the WS will be no longer available. To implement this, the WS test infrastructure needs to keep track the time the supplied WS can be accessed.

Access restriction: Instead of tracking time or usage, the UDDI server may provide those WS that can be involved by the test scripts supplied by the UDDI server only. In

this way, a client can invoke the WS only via the supplied test scripts, but not directly. In this approach, the WS test infrastructure does not need to track the time or count, however, each test script supplied must be carefully designed so that they cannot be compromised. For example, the test scripts supplied can have a digital signature and only signed test scripts can access the WS supplied during the intermediate checkout process. Note that using digital signature for executable test scripts does not prevent the UDDI server to release the source code for the client. The UDDI server can release both the source code and executable test code to a client, but only the executable will be signed. While the WS test infrastructure will be simpler for this approach, this approach reduces the confidence for a client with respect to the supplied test script because the client is not able to construct new test scripts to customize the verification.

Hybrid approach: The UDDI server may supply multiple approaches to this problem for different WS providers and clients. The advantage of this approach is that different strategies can be used to meet the different needs of WS providers and clients, but the test infrastructure will be more complex.

Another important consideration is that a WS provider may see different test scripts than a WS client. This is needed because a particular WS may need to address its specific check-in process while some of issues involved in the check-in may not be needed for the checkout. For example, a WS provider may need to open up some of its source code for security testing and thus the WS will be subject to more check-in verification than checkout verification, as the client will be able to see the executable version of the WS only and thus the client will not see those verification scripts involved in source code analysis. In other words, the UDDI server may need to distinguish those test scripts that needed for both check-in and checkout, and those only for check-in only.

3. Test Scripts for Testing WS

This section addresses the kinds of test scripts that should be attached to a UDDI server. This depends on the kinds of assurance needed and the application of the associated WS. In general, the following five kinds of test scripts are needed:

Method testing scripts: these are scripts that aimed at testing methods provided by WS.

Object testing scripts: these are scripts that aimed at testing the WS as an individual object only, i.e., without consideration to other WS that are involved in an application.

Integration testing scripts: these are scripts that aimed at testing the WS for an integrated application that involved multiple WS.

System integration testing scripts: These are scripts used to ensure that the system performs the intended function.

Domain testing scripts: These are test scripts that apply to all the WS for a given domain or sub-domain.

A WS provider may wish to provide those method and object testing scripts so to increase for their clients' confidence in using the supplied WS. The WS provider may wish to supply a variety of test scripts that address different issues of the WS, e.g., functional, robustness, reliability performance, and scalability aspects. For example, a FFT (Fast Fourier Transform) WS provider may wish to provide test scripts that supply inputs that are easy to identify their expected outputs because in general it is difficult to know whether the generated output is correct for an arbitrary input without another FFT routine or table at hand; however, it is relatively easy to determine the FFT for certain inputs.

A WS provider may want to provide test scripts that supply incorrect input so to demonstrate the *robustness* of WS. Another WS provider may provide test scripts that provide test coverage according to the typical operational profile of WS to demonstrate the *reliability* of the WS. Another WS provider may supply a test script that allows its clients to specify various input size to determine the *performance* and *scalability* of the WS under test. For example, a sorting WS provider may allow clients to specify input size ranging from few data items to millions of items. Once a WS provider gathers sufficient user feedback, the provider will be able to supply competitive test scripts for most future clients for the particular WS under test.

3.1. Services Integration Test Scripts

While method and object test scripts are sufficient for testing small WS, for many applications, WS need to collaborate with other WS to perform the mission. A unique feature of WS is that they can be selected, integrated and called by unknown parties at runtime. Several OO system integration specification techniques are available, e.g., method sequence specification MtSS [5][6], message sequence specification MgSS, and framework sequence specification MfSS [10]. MfSS specifies the sequence constraints on the interaction between framework's objects and custom's objects (or application objects) [4]. In addition, MfSS also specifies dynamic typing and dynamic binding. MfSS involves with multiple objects, thus a sequence expression must

provide object name together with its method name using Object Constraint Language (OCL) [20]. These specifications can be added into WSDL so that a client can use to get the best match during the UDDI search.

3.2. System Integration Testing Scripts

This kind of testing scripts is often derived from system requirements. For example, in a supply-chain management (SCM) system, participants include *suppliers*, *manufacturers*, *wholesalers*, *retailers*, and *customers* [21], and each can be a WS. For example, a *retailer* sells products to *consumers*. A *retailer* manages its *consumers'* information including payment details and addresses, so the *consumer* does not need to provide the payment information at the purchase time. A *retailer* replenishes its *warehouses* when the inventory drops to a certain level. For example, scenario "*Customer Successfully Purchases Goods*" is a system scenario reflecting interaction among *customer*, *retailer*, and *manufacturer* WS because a typical scenario consists of "*A customer accesses to the retailer system with customer ID*", and "*purchases the product by specifying the product and quantity*", "*the retailer ships the requested product, replenish the products by placing order to the manufacturer due to the inventory dropping close to the minimum amount*".

3.3. Domain Testing Scripts

This is similar to system testing scripts, however, a domain testing script is applicable to all the systems that belong to the domain. Many domains have a set of common workflows, and the common workflows can be used for specifying system scenarios for all the systems in a particular domain [7]. For example, Web Service Interoperability (WS-I) organization defines standard SCM use cases and usage scenarios [21]. The specification also states interactions between participants. For example, scenario "*Customer successfully purchases goods*" is a domain scenario between a *customer* and a *retailer*. For a WS to be added to a UDDI registry, the WS provider needs to supply test scripts for method testing scripts, object testing scripts, and probably system testing scripts, but the WS will be verified by domain testing scripts for all the sub-domains of the WS declares that it belongs to.

3.4. Test Scripts for Sub-Domains

Once scenarios for a domain are available, it is possible to develop scenarios for its sub-domains by including additional constraints among participants. For example, domain scenarios for food SCM can be developed by adding timing constraints to the general SCM scenarios to ensure the food shipped is fresh. Similar to the OO technology, domain scenarios for a domain and its sub-domains can be organized in a hierarchical manner (Figure 5) with scenarios specified in a parent domain applicable to its sub-domains, but not the other way around. Domain information, domain scenario information and test scripts information have relation with each other, and the Factory design pattern [3] can be used to create the scenario objects and test scripts objects regarding to that domain as shown in Figure 5.

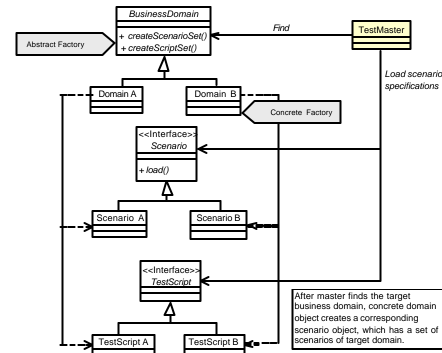

Figure 5. Domain, Scenario, and Scripts Hierarchy

An important issue is that all the related test scripts must ensure the associated WS interoperate with each other. For example, as illustrated in Figure 6, a *retailer* domain can have a *food retailer* sub-domain, and in turn has a *specialty food retailer* sub-domain; and similarly a *manufacturer* has a *food manufacturer* sub-domain, and in turn has a *specialty food manufacturer*. A *retailer* WS should be able to interoperate with the entire *manufacturer* WS and *food manufacturer* WS, as well as the entire *specialty food manufacturer* WS from all the WS providers. Similarly, a food manufacturer must be able to interoperate the entire *retailer* WS, *food retailer* WS, and *specialty food retailer* WS. Due to these constraints, test scripts for a particular WS cannot be arbitrarily developed without consideration of other WS. Specifically, the test scripts for *retailer*s cannot be independently developed without considering *manufacturer* WS, and vice versa, and this applies to the test scripts for all of their sub-domains too.

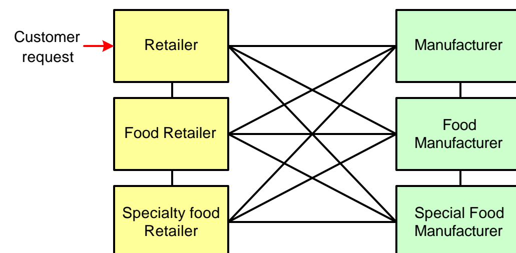

Figure 6. Interoperability

This kind of complex relationship can be expressed using MfSS. For example, the MfSS of "Customer successfully purchases products" can be expressed as follows:

$$CustomerLogin_{ws_info} \cdot RetailerShipProduct_{WS_info} \cdot CustomerPayment_{WS_info}$$

where *WS_info* refers to the WS information. Here, the customer can be a regular retail *customer*, a *corporate customer*; the retailer can be a regular *retailer*, a *food retailer*, or a *specialty food retailer*; the manufacturer can be a regular *manufacturer*, a *food manufacturer*, or a *specialty food manufacturer*. Regardless whether the customer is a regular *customer* or a *corporate customer*, it must follow the MfSS specification and constraints, otherwise *customer* or *corporate customer* WS will not be able to interoperate with other WS such as *manufacturers* or *retailers* WS.

3.5. Evolution of Test Scripts during Domain Evolution

Another important issue is that as systems keep on changing, the corresponding test scripts must keep on updated. When a new sub-domain is created, it is important that all the test scripts from its parent domain are inherited with possibly additional constraints to ensure interoperability. For example, a typical test script often follows a three-step process: *setup*, *execute*, and *verify*, and this three-step process can be inherited from the parent domain to its sub-domain using Template Method design pattern [3]. By using this design pattern, a test script in the sub-domain can define its own methods while maintaining the overall structure of the test script from the parent domain.

Furthermore, a tester can create complex test scripts by reusing existing test scripts for integration testing and stress testing using the Composite design pattern (Figure 7) [3]. The Composite design pattern treats individual objects and their compositions uniformly. A tester can use different strategies to compose the existing test scripts, and use the Strategy design pattern to encapsulate this. Strategy pattern allows the user to easily add new strategies without changing the structure. The user can use four operators to composite test scripts together: sequencing, looping, conditioning, and concurrent. For example, simple WS "*A customer successfully login to a retailer*" and "*A retailer sends a product catalog to a customer*" can be composed using "concurrent" and "sequencing" operators to generate a test script: "*multiple customers login to a retailer to request the product catalog*", which can be used to perform stress testing on the *retailer* WS.

Figure 7. Test Script Generator

4. Web Services Test Execution

Test execution is performed with three components: test master, test agents, and test monitors. They can be at same site, but often they are distributed to different sites. Test monitors often reside at the same site as the WS under test. These three components are often located at different sites, and communicate with each other using SOAP (Figure 8) [16].

Figure 8. Test Master, Agents and Monitors

A test master manages testing by creating scenarios from the database, generating test scripts, and running test by sending commands to one or more agents concurrently using SOAP. It also receives data from one or more test monitors either synchronously or asynchronously.

Test agents act as proxies of the test master with respect to the WS, execute tests, verify the results, and report results to the test master.

Test monitors capture the exchange messages among WS and their clients, and record any state changes. A test agent can also act as a test monitor if it resides at the same site as the WS under test.

WSDL files that contain the signatures specification of all the web services methods including method names, and input/output parameters [14][17]. WSDL can be extended so that a variety of test techniques can be used to generate test cases [12]. The test master extracts the interface information from the WSDL file and maps the signatures of web services into test scenarios. The test cases are generated from the test scenarios in the XML format that is interpreted by test agents for execution and verification [16][22].

The test master participates in check-in and checkout of UDDI (Figure 9):

- Looks up the domain scenario specification (*findBusinessDomain*()) according to domain information of the WS provider described in *CategoryBag* object;
- Generates different levels of test scripts (*TestScript*) from scenarios (See Figure 5 and Figure 9):
    - Method test scripts
    - Object test scripts
    - Integration test scripts
    - System test scripts
    - Domain test scripts
- Initiates the testing by sending commands to test agents (*runScenario()*) using SOAP;
- Collects test result reports from *TestAgent* and messages from *Monitor* for testing analyses;
- Verifies interaction patterns identified among systems.

The test master also manages the dependency and interaction among participants.

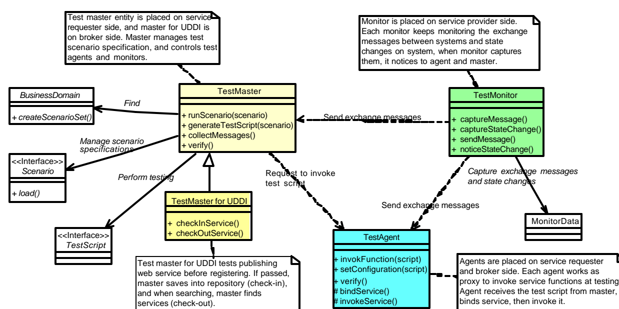

Figure 9. Class Diagram of Test Agent, Master Monitor

Test agents provide the following functions:

?? Executes remote testing: This is done by binding and invoking appropriate methods of the participating WS. If more than one WS are involved, test agents need to send commands to all the participating WS for execution.

?? Verifies the test results against expected outputs.

Test monitor provides the following functions:

?? Monitors the exchange messages with timestamps among clients and service providers (*captureMessage()*);

?? Traces state changes of the systems (*captureStateChange()*) with respect to requests. With the trace information, the systems can roll back for unfinished transactions if needed. This feature is useful for database transaction management.

?? Keeps track of asynchronous messages, notices the clients (*TestAgent*) when the asynchronous requests have arrived (*NoticeStateChange()*).

?? Sends the exchange information to the test master and test agents that are listed as observers of the monitor.

For WS testing, the master, agents and monitors interact and collaborate with each other:

1. A tester specifies one or a group of scenarios, and starts running it;
2. *Test master* loads the scenario data using *Scenario* class;
3. *Test master* generates the *Test Script* with test cases from the loaded scenario data.
4. *Test master* requests *Test Agent* to execute the system function or set configuration of system following the steps of *Test Script*.
5. *Test agent* binds and invokes the system function/service.
6. Each *Test monitor* keeps monitoring the associated systems, and it captures the exchange messages among systems and state changes of systems.
7. *Test monitor* captures the information and sends them back to *Test Master* and *Test Agent*.
8. *Test master* collects the test results, and verifies the test results including interactions between the participating WS.

5. Distributed WS Testing on the Web

To improve the verification process by the UDDI server, one can use free computing resources on the web to run test agents. Currently almost four hundred millions of computers are connected to the Internet, even 0.1% of them agree to help, and 400,000 computers will be available [1]. To do this, one needs two more components: Registry of Volunteer-Computers (RVC) and Testing Scheduler (TS). The volunteer computers register to the test master and get the software for test agents installed. The test agents send free resource information to the test master including machine type, process capability, and available time. RVC maintains a table of available resources.

The test master sends scheduling request to the TS, and it does the following tasks:

?? Estimates testing execution time;

?? Rank the requests according to the priorities of requests;

?? Matches the expected testing execution time and the resource available in the RVC; and

?? Send the scheduling results to the test master.

The test master then initiates testing by sending commands to the test agents on the volunteer computers according to the scheduling results. This approach can be used to see how the behavior of WS under stress testing where numerous test agents can initiate the testing simultaneously.

This mechanism has been implemented and Figure 10 shows the actions of test agents. With the assistance of these volunteer computers, it is possible to perform significant verification of WS at a low cost and allow the UDDI server to spend its resource on other activities instead of on verification of WS.

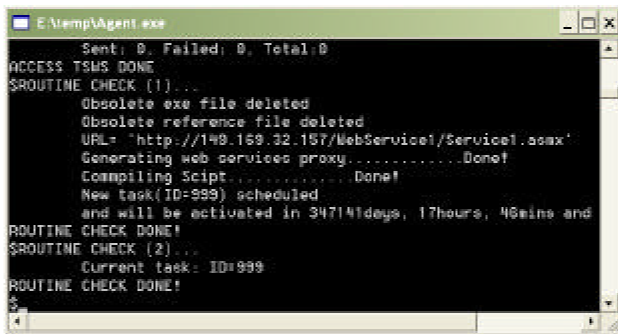

Figure 10. Volunteer Computer Agent

6. Conclusion

To improve the assurance of WS, this paper proposes adding verification features to the UDDI server, specifically, the check-in and checkout mechanisms. The check-in process requires WS to pass various tests before they can be registered at the UDDI registry; the checkout process allows a clients to test drive WS before checking out from the registry. A WS provider may also supply additional test scripts that a client can use to test the WS before application. Furthermore, this paper proposes to organize test scripts in a hierarchical manner like the OO inheritance tree. In this way, test scripts can be evolved with the evolution of WS, and the interoperability of test scripts can be systematically addressed. This paper also proposed a distributed test execution mechanism, including test master, test agents, and test monitors to test distributed WS at different sites. Finally, this paper proposed using free computer resources that are available on the web to perform WS verification to reduce the burden of the UDDI server.

References

[1] D. Arnow, G. Weiss, K. Ying, and D. Clark, "SWC: A Small Framework for WebComputing", http://www.parco99.tudelft.nl/content.html

[2] X. Bai, W. T. Tsai, R. Paul, K. Feng, and L. Yu, "Scenario-Based Modeling and Its Applications to Object-Oriented Analysis, Design, and Testing", Proc. of IEEE WORDS 2002, pp. 140-151.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Paterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, MA, 1994.

[4] M. Fayad, D. C. Schmidt, and R. E. Johnson, *Building Application Frameworks*, Wiley, New York, NY, 1999.

[5] S. Kirani and W. T. Tsai, "Method Sequence Specification and Verification of Classes", Journal of Object-Oriented Programming, Oct. 1994, pp. 28-38.

[6] S. Kirani and W. T. Tsai, "Specification and Verification of Object-Oriented Programs", Technical Report TR94-64, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, 1994.

[7] NAICS: North American Industry Classification System, http://www.census.gov/epcd/www/naics.html

[8] OASIS, Universal Description, Discovery, and Integration (UDDI), http://www.uddi.org

[9] R. Paul, W. T. Tsai, and L. Yu, "Rapid and Adaptive E2E Test and Evaluation of SOS", submitted to STC, 2003.

[10] W. T. Tsai, Y. Tu, W. Shao and E. Ebner, "Testing Extensible Design Patterns in Object-Oriented Frameworks through Hierarchical Scenario Templates", Proc. of IEEE COMPSAC, 1999, pp. 166-171.

[11] W.T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-End Integration Testing Design", Proc. of IEEE COMPSAC, 2001, pp. 166-171.

[12] W. T. Tsai, L. Yu, R. Paul, T. Liu, and A. Saimi, "Developing Adaptive Test Frameworks for Testing State-Based Embedded Systems", in Proc. of IDPT, 2002.

[13] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, "Extending WSDL to Facilitate Web Services Testing", Proc. of IEEE HASE, 2002, pp. 171-172.

[14] W. T. Tsai, R. Paul, W. Song, and Z. Cao, "Coyote – Object-Oriented Web Service Testing Framework", Proc. of IEEE HASE, 2002, pp. 173-174.

[15] W. T. Tsai, L. Yu, X. Liu, A. Saimi, and Y. Xiao, "Scenario-Based Test Generation Tool for Embedded Systems", to appear in Proc. of IEEE IPCCC 2003.

[16] W. T. Tsai, L. Yu, A. Saimi, and R. Paul, "Scenario-Based Object-Oriented Test Frameworks for Testing Distributed Systems", to appear in Proc. of IEEE Future Trend of Distributed Computing Systems, 2003.

[17] UDDI Version 2.0 Data Structure Reference. http://www.uddi.org/pubs/DataStructure-V2.00-Open-20010608.pdf.

[18] UDDI Version 3.0, "UDDI Version 3 Features List", http://www.uddi.org/pubs/uddi_v3_features.htm

[19] W3C, Web Services Definition Language http://www.w3.org/TR/wsdl.

[20] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, Reading, MA, 1999.

[21] WS-I: http://www.ws-i.org/, "WS-I Usage Scenarios", "Supply Chain Management Use Case Model", "Sample Application Supply Chain Management Architecture".

[22] F. Zhu, "A Requirement Verification Framework for Real-time Embedded Systems", PhD dissertation, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, 2002.