Supporting Dynamic Changes in Web Service Environments

Mohammad Salman Akram, Brahim Medjahed, and Athman Bouguettaya

Department of Computer Science, Virginia Tech
7054 Haycock Road, Falls Church VA 22043, USA
{salman,brahim,athman}@vt.edu

Abstract. The Web has become the universal medium for publishing and using of Web accessible services called *Web services*. The widespread adoption of XML standards including WSDL, SOAP, and UDDI has spurred an intense research activity to deal with issues related to Web services. One of the most important issues is the management of changes that occur in Web service environments. Web services operate in a highly dynamic environment where changes can be initiated to adapt to evolving business climates. All changes performed to Web services must be efficiently propagated to ensure global consistency. In this paper, we combine *Web services*, *ontologies*, and *agents* to cater for the management of changes in Web services. We address the challenging issues of *detection*, *propagation*, and *reaction* to both *internal* and *external* changes to Web services.

1 Introduction

The role of the Web is shifting from a distributed information storage to a world wide service provider. This shift is being propelled by the current work on Web service standards, both in industry and academia. A Web service is a set of related functionalities that can be programmatically invoked through the Web [1]. The Web service framework facilitates dynamic and efficient methods of interactions on the Web. Web services are poised to be the foundation of the envisioned service oriented architecture [2].

The Web service model involves three types of participants: *requester*, *provider*, and *registry* [3]. In a simple scenario, the service provider first publishes its WSDL (Web Service Description Language) description in a UDDI (Universal Description, Discovery, and Integration) [4]. A service requester then searches the UDDI for a Web service that matches a given criteria. If the search is successful (i.e. one or more service providers are located), the service requester invokes the service provider using SOAP (Simple Object Access Protocol) messaging. This depicts a simplified model of service request processing. The ultimate goal of Web services is to serve as independent components in loosely coupled systems such as electronic marketplaces (or e-marketplaces) [5] and Web based Virtual Enterprises [6]. These systems typically process requests that target more than one Web service. Service requests in these systems become significantly more dynamic and complex as the number of target services increases [7]. The success of fulfilling such service requests relies on dealing with the *volatile* and highly *dynamic* nature of Web services. Additionally, the service request must interact with an *exploratory* service space for locating Web services. We characterize the Web service environment by the following features:

– **Exploratory:** Exploratory refers to the *nondeterministic* process of identifying Web services necessary for a given request. Web services are *a priori* unknown and may only be determined dynamically, i.e., after the need for the service is established. It is not required that the interacting entities be cognizant of each other prior to interaction.
– **Volatile:** Volatility implies that a Web service answering a request at any given time may not be available to answer the same request at a later time. Once provider services are selected, it is possible that those services may be inaccessible before or during the execution of a request.
– **Dynamic:** Web services have highly dynamic content. The *content* of a Web service refers to the information it provides through its operations (e.g., the price of a given stock. This content may change frequently and unpredictably. Furthermore, changes may occur while requests are being processed and affect the overall execution of a request.

In this paper, we describe an architecture that supports requests over exploratory, volatile, and highly dynamic Web services. We propose the use of *ontologies* to select services from an exploratory service space [8,9,10]. We employ *agents* to deal with Web service volatility and dynamism. This will be accomplished by managing change to Web services [6]. Our approach is illustrated using an electronic stock market. In this example, we simulate user requests to buy, sell, and inquire about stocks of one or more companies. The environment where these requests are executed is characterized by three features: (i) Web services that provide interaction with stock markets are *a priori* unknown, (ii) Web services are volatile and their availability is uncertain, and (iii) the content (e.g. stock prices) of the Web services is highly dynamic (i.e., changes very frequently).

The paper is organized as follows: In Section 2, we propose a model for dynamic service requests. Section 3 is a detailed description of the proposed conceptual architecture. In Section 4, we present the implementation of this architecture within the context of an electronic stock market. We provide related work in section 5. Finally, we provide some concluding remarks in Section 6.

2 A Model for Dynamic Web Service Change Management

Servicing requests is a challenging issue when targeting environments consisting of volatile and dynamic Web services. The execution of service requests must be supported by the (i) discovery and selection of Web services in an exploratory service space and (ii) an environment that adapts to changes to Web services. Changes to Web services may be internal or external. *Internal* change refers the dynamic nature of Web service content. Internal change includes change in the information provided by a Web service. *External* change refers to the volatility of the Web service. This type of change includes the unavailability of a service and its operations during the execution of a request. The problem of supporting dynamic service requests has two facets:

– **Web service discovery and selection:** An important step in the process of answering a service request is to discover and select Web services with appropriate functionality. A request may involve several distributed and remote services that are determined on the fly (i.e., at run-time). Service discovery and selection may be initiated when the request is first issued or it may be a result of change. The challenge is to dynamically and efficiently discover the appropriate services for a given request within an acceptable response time.
– **Adapting to Web service changes:** Any request dependent on a Web service must deal with the issue of change management. *Change management* is the timely detection, propagation, and reaction to both internal and external changes. *Detection* is the method of identifying a change that is of interest to the system. *Propagation* requires that all interested system components be informed of those changes. *Reaction* is initiating a compensatory process in response to a change.

2.1 Service Request Specification

The service space in our proposed model consists of a set of remote Web services registered with a global service registry. Each Web service provides some stock market functionality through its operations. In particular, it provides information about the stocks available in that market and the ability to trade them. In our scenario, Web services provide access to stock markets located in New York (NYSE), London (FTSE), Paris (CAC), and Tokyo (TSE). Each Web service specializes in one or more stock categories. Examples of categories include bookstore, Internet service provider (ISP), and travel.

A *service request* is a list of atomic (sequential and concurrent) orders. Each *order* is defined by the order type (`type`), stock identifier (`stockID`), stock category (`stockCatg`), stock market (`stockMarket`), acceptable price (`price`), number of shares (`quantity`), and the duration of the order (`time`). A request may consist of one or more of these orders. Furthermore, a single order may be divided into one or more execution threads. An *execution thread* is an instance of an order interacting with a single Web service. Formally, a request may be represented by any expression generated by the following language:

```
Req ::=    Req ; Req |
           Req || Req |
           Order
```

Where:

- ";" denotes a set of two orders that must take place in the given sequence
- "||" denotes the concurrent (i.e., parallel) execution of two orders
- *Order* is defined as the 7-uple:
  `(type, stockID, stockCatg, stockMarket, price, numShares, time)`

Example:

```
Req1:    { ("sell", AMZN, Bookstore, NYSE, $21, 500, 00:30) ||
           ("sell", YHOO, ISP, NYSE, $19, 350, 00:30) ;
           ("buy", NTC, Computer, CAC, $16.5, 950, 01:00) }
```

The previous request translates to concurrently attempt to sell 500 shares of *Amazon.com* (AMZN) at a price not less than \$21 and 350 shares of *Yahoo!* (YHOO) at a price not less than \$19 per share. The orders should be repeatedly attempted until they are terminated by (i) a successful execution or (ii) the thirty minute time limit (whichever comes first). If and only if the two previous orders succeed, then attempt to buy 950 shares of *Intel Corporation* (NTC) at a price not more than \$16.5 per share. This order would be attempted for a one hour interval.

Req$_1$ provides an example of "buy" and "sell" order types. A user may also issue an "inquiry" request that asks for information about a particular stock or a stock category. Req$_2$ is an example of a request that inquires about the values of all the stocks in the *ISP* category at the NYSE. The inquiry will continue for a period of one hour. The highest and lowest prices of each stock in the *ISP* category will be reported to the user.

```
Req2:    { ("inquiry", *, ISP, NYSE, *, *, 01:00) }
```

The asterix (*) represents an undefined value for the corresponding attribute. For example, the first asterix implies that the `stockID` is unknown. The use of asterix in service requests introduces flexibility and dynamism in user requirements. Service request may also target multiple stock markets. Req$_3$ illustrates a situation where a user is searching for the stock market that currently offers NTC shares at \$16 or lower. Notice that the asterix in the time attribute denotes immediate execution of the request.

```
Req3:    { ("inquiry", NTC, *, *, $16, 220, *) }
```

The examples above indicate that simple user requests may become extremely complex at processing time. Figure 1 displays the process of resolving a service request. The diamonds indicate the stages in request processing. Solid arrows describe the control flow between stages. Dotted arrows represent the access and update of service descriptions. All requests issued by the user are first parsed to validate the format of the request. A parsed request is decomposed into atomic orders, if needed. For example, Req$_1$ illustrates situations where a request is translated into three orders. Each order is evaluated to determine the required

Web services. For instance, Req_3 implies inquiry for NTC stocks in all available services. In this case, the order will be divided into several execution threads with each thread mapping to a single Web service. After service invocation, the request enters the change management stage. If the execution is successful, a response is generated and sent back to the user. In the next subsection, we describe the use of ontologies for the service selection stage.

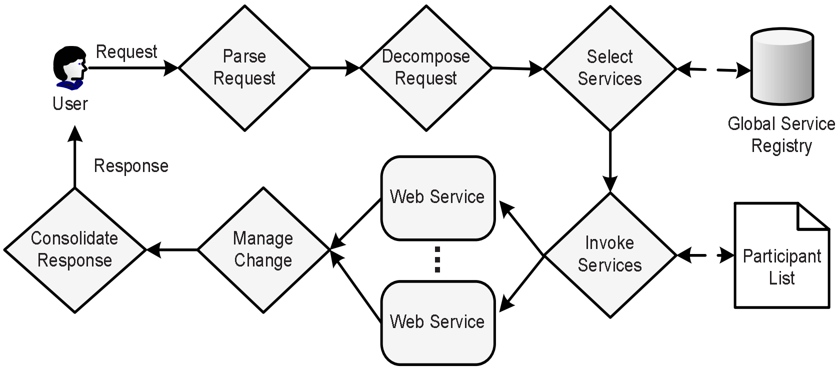

Fig. 1. Request Processing

2.2 Using Ontologies for Locating Web Services

The problem of discovering and selecting the services necessary to answer requests is particularly challenging in the dynamic Web context. Discovering appropriate Web services dynamically is essential to answering a service request (Figure 1). Our approach to the problem is based on the idea of organizing Web services into ontologies. An *ontology* describes a coherent slice of the service space, i.e., a collection of Web services that share the same domain of interest [11]. In our stock market example, the category of a Web service defines its domain of interest. For example, services that deal with AMZN stocks may belong to a *bookstore* ontology.

DAML-S (DARPA Agent Markup Language for Web services) provides the ability to organize Web services into ontologies [12]. DAML-S divides service descriptions into the service profile, model, and grounding. The service *profile* provides a high level description of a Web service. It expresses required input of the service and the output the service will provide to the requester. The service *model* defines the operations and their execution flow in the Web service. Service *grounding* provides a mapping between DAML-S and the WSDL standard and describes how the service may actually be invoked.

The service profile provides sufficient information for discovery. It is divided into a description of the service, the functionalities, and the functional attributes.

The *description* provides human understandable information about the Web service. For example, a description includes the name of the service and its textual description. A *functionality* in DAML-S describes properties like input, output, precondition, effect, etc. The *functional attributes* provide information such as response time and costs [13,12].

To understand how DAML-S may be used in our solution, let us refer to our stock market example. Stock markets generally involve many categories of businesses. Each Web service may belong to one or more categories. We define these different categories for our Web services using DAML-S. Figure 2 depicts an excerpt of a service profile description for our stock market example using DAML-S.

```
(1)  <daml:Class rdf:ID="NYSE">
(2)    <rdfs:label>NewYorkStockExchange</rdfs:label>
(3)    <rdfs:subClassOf rdf:resource="&service;"/>
(4)  <daml:Class>

(5)  <rdf:Property rdf:ID="Bookstore">
(6)    <rdfs:label>Bookstore</rdfs:label>
(7)    <rdfs:subPropertyOf rdf:resource="&profile;serviceCategory"/>
(8)    <rdfs:domain rdf:resource="&service;serviceProfile"/>
(9)    <rdfs:range rdf:resource="&daml;#Thing"/>
(10) </rdf:Property>

(11) <rdf:Property rdf:ID="Travel">
(12)   <rdfs:label>Travel</rdfs:label>
(13)   <rdfs:subPropertyOf rdf:resource="&profile;serviceCategory"/>
(14)   <rdfs:domain rdf:resource="&service;serviceProfile"/>
(15)   <rdfs:range rdf:resource="&daml;#Thing"/>
(16) </rdf:Property>

(17) <rdf:Property rdf:ID="ISP">
(18)   <rdfs:label>ISP</rdfs:label>
(19)   <rdfs:subPropertyOf rdf:resource="&profile;serviceCategory"/>
(20)   <rdfs:domain rdf:resource="&service;serviceProfile"/>
(21)   <rdfs:range rdf:resource="&daml;#Thing"/>
(22) </rdf:Property>
```

Fig. 2. DAML-S Description

This description is for the Web service that provides interactions with NYSE. Lines 1-4 present the human understandable description of the service profile. It defines the class name (NYSE) and also indicates that the class is a *service*. The remaining lines define categories of the service. In this case, the Web service belongs to three categories. Lines 5-10 define a service property that implies the Web service deals with stocks in the bookstore category. Lines 11-16 indicate the Web service belongs to the travel ontology. Finally, lines 17-22 describe the Web service's membership in the ISP category.

Web services needed to answer a request are discovered using descriptions provided in DAML-S registries. The selection criteria for the service is first extracted from the order. For example, Req_2 needs all Web services from the *ISP* category that provide services for NYSE. In this case, a search is initiated in the DAML-S registry for a class name that matches "NYSE". If such a class name is located, the search will continue to check the properties of the Web service. The service property must match *ISP*. The fulfillment of these steps indicate that a Web service has been located successfully. After a service has been located, it may be invoked by using the service *grounding* in DAML-S and its mapping to WSDL [12].

Individual services may join or leave the formed ontologies at their own discretion. An overlap of two ontologies implies that a service provides information that is of interest to both ontological domains. For example, the intersection between the two ontologies *bookstore* and *ISP* may contain services that are involved in both types of companies.

2.3 Managing Changes

Requests targeting service providers require mechanisms to *detect*, *propagate*, and *react* to changes in the provider services. Changes to Web services can be planned or unexpected. They may occur in several forms. We classify the changes that need to be managed in a request as:

– **External:** External change primarily refers to temporary or permanent unavailability of a service or its operations. This unavailability may occur before or during the execution of the request. It may be a result of a network failure, service relocation, request overload, operation rename, etc. In this case, the request needs to be rolled back and an alternate service must be selected to fulfill the request.
– **Internal:** Any change in the content of a Web service may need to be detected to fulfill a request constraint. In our example of stock markets, the values for stocks change very frequently. Important (financial) consequences may result if the mechanism handling change management fails to react promptly to fluctuations in the markets. These reactions may only be required if the change reaches a certain threshold [14].

Consider the following request:

```
Req4:    { ("buy", AMZN, *, *, *, 150, 00:15) }
```

Req_4 translates into an order to buy 150 shares of AMZN (from any market) with the lowest price in the interval of fifteen minutes. Assume that while the system is executing this order at NYSE (because the lowest price for AMZN was quoted there), a change occurred at CAC that made AMZN's price fall below its price at New York. To react to this change and fulfill the user's constraint (i.e., buying at the lowest price), the execution of purchase order at NYSE must be aborted and performed at CAC. In the previous example, the replacement was not the result of a failure but, rather, the result of a constraint violation. Using the same request, let us assume that while a service request is executing at the CAC Web service, a network failure occurs and the service becomes unavailable. In order to fulfill the request, the execution at the respective CAC Web service needs to be rolled back and an alternate service (that meets given constraints) should be invoked.

The examples illustrate a need for change management in a Web service environment. Change management consists of detection, propagation, and reaction to changes. Our approach to solving this problem is based on using agents that play the role of *monitors* and *notifiers* [15,6]. These agents are background processes that monitor the participant Web services for relevant changes (e.g. changes in stock price, unavailability of Web service) and notify entities concerned with the change.

Change detection: Change detection is the awareness that a change has occurred and the subsequent identification of its cause. It deals with changes that occur (i) before a provider service is invoked and (ii) during execution of a Web service. For changes that occur before service invocation, we employ change detection through soft states [16]. *Soft states* is a method used to maintain membership of entities in a loosely coupled system. This method requires that a member periodically send "refresh" messages to renew its membership. These messages are sent to a node that maintains the membership list.

In our case, the provider Web services are members or participants in the loosely coupled system. The membership information is stored in the *participant list*. A participant list is generated for every request. Agents act as intermediaries between the participant list and the Web services. They also maintain the participant list by updating the list at time of change. An participating Web service is assigned to each agent that monitors changes in the status of that service. This agent periodically verifies the availability of the service and its operations, and the contents of the Web service [14]. To verify changes in the availability of a service, the agent will send "alive" messages to the Web service. If the Web service responds, the service is assumed to be available. However, if a response message is not received from the Web service within an acceptable time limit, the service is considered as unavailable.

Changes to operations are detected by retrieving the service descriptions from the UDDI. Any change to a service operation (e.g., rename, change of parameters, etc.) implies that the change was made explicitly by the Web service programmers. This justifies our assumption that the Web service description in the UDDI will be appropriately updated after an operation change. Change in content of a Web service requires the assigned agent to repeatedly invoke an operation that provides the respective content. Let us take the example where a service request is inquiring for the price of AMZN stocks in the next half hour.

The agent will invoke the operation in regular intervals (e.g., thirty seconds) and compare output values with values provided in the previous invocation.

Change that occurs during an interaction with a Web service is detected by the failure of service invocation. The reason of the failure is identified by the agent that manages the particular Web service. The technique for identifying the cause consists of change detection through the process mentioned above. For all changes that are detected, the agent uses appropriate mechanisms for propagation and reaction as explained below.

Change propagation: We use the participant list to propagate changes in the system. The participant list contains references to all Web services that have been selected to answer the service request (i.e. that are part of the system). Agents will update the participant list in case of any external change. The update involves the removal of the service reference stored in the participant list. Since a service reference must be present in the participant list before it can be invoked, the removal of service reference terminates the membership of that service. If an internal change (i.e. change in the data provided) is detected, the data used for response consolidation is updated to reflect the change.

Reaction to changes: Reaction to change depends on the (i) type of change and (ii) availability of alternate services. In case of an external change, an alternate service must be selected to fulfill the user request. The service selection stage is initiated and provided with the description of the required service. If an appropriate service does not exist (or cannot be located), the request must be canceled. However, if an alternate service is selected successfully, it is registered with the participant list and request processing is resumed. In the event of an internal change (e.g. change in stock prices), the previous data will be replaced with the current response.

Change management algorithm: Figure 3 displays an algorithm that describes the change management process. The algorithm takes two parameters: `request_time` and `participant_list`. `request_time` is the `time` attribute extracted from the order. `participant_list` contains the list of all Web services that are currently participating in the system. The algorithm executes for the duration of the request. The algorithm starts by verifying the availability of each Web service by sending alive messages to the Web services in the the `participant_list`. If a Web service is not available, it is removed from the `participant_list` and an alternate service is selected. Second, the algorithm requires checking for changes in operations. The respective agent retrieves the current description of each Web service from the global service registry and compares it with the description in the system. If the description has changed, the agent removes the service description from the `participant_list` and selects an alternate service. Finally, the agent compares the contents returned by the Web service with the contents of the previous response. If the agent detects any change, the response must be reconsolidated.

```
Input: request_time, participant_list
{
 time = request_time
 while (request_time != 0)
 {
  for each Web Service WS in participant_list
  {
   send alive message to WS
   if not alive then
   {
    remove WS from participant_list
    call (Service_Selection (service_description (WS)))
   break
   }

   global_description = WS service_description from global_service_registry
   if service_description (WS) not equals global_description
   {
    remove WS from participant_list
    call (Service_Selection (service_description (WS)))
   break
   }
   current_data = invoke WS.operation
   if current_data not equals previous_data
   {
    call (Response_Consolidation (current_Data))
    break
   }
  }
  decrement time
 }
}
```

Fig. 3. Change Management Algorithm

3 Proposed Architecture

The architecture proposed to support dynamic service requests is illustrated through the electronic stock market example. A Graphical User Interface (GUI) is used to interact with the user. A *request broker* serves as an intermediary between the GUI and the participant Web services. The concept of a request broker is very similar to a stock broker in a stock market and it serves as an integral part of our architecture.

In our proposed architecture, the GUI allows users to specify and submit their requests (Figure 4). It supports both short-lived requests (e.g., inquiry about the current value of a given stock) and long-lived requests (e.g., inquiry about the lowest value of a given stock in one hour). The GUI is also responsible for parsing and validating requests and sending them to the Request Broker. The Request Broker (RB) implements the remaining stages of request processing (Figure 1). Specifically, it performs decomposition, service selection, service invocation, change management, and response consolidation.

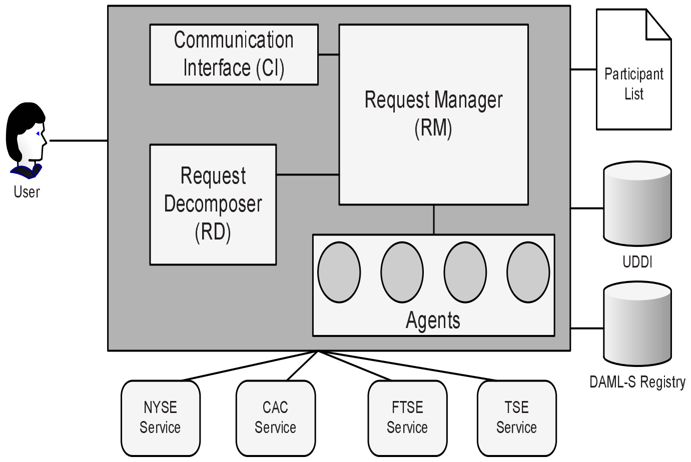

Fig. 4. Architecture

Every user is assigned a single instance of the RB. Communication between RBs and the various registered Web services takes place using the SOAP standard (Simple Object Access Protocol). Each Web service corresponds to one stock market (e.g. NYSE) and provides information about the stocks associated with the stock market. In the remainder of this section, we first elaborate on the functionality of the RB with reference to our stock market example and then present the internal components of the RB.

3.1 Request Broker Functionality

Request Brokers are the intermediaries between the GUI and service providers. Each RB deals with requests from one particular user. In essence, the RB is responsible for four major functions:

– **Request Decomposition:** When an RB receives a request from the user (via GUI), it translates that request into one or more orders. These orders are further broken down into execution threads that invoke operations at the underlying Web services or *participants*. Consider the following request:

  $\texttt{Req}_5$: `{ ("inquiry", AMZN, *, *, *, *, *) }`

  When the RB receives the order corresponding to this request, it generates multiple SOAP messages, each invoking a Web service that represents stock markets dealing with *AMZN* stocks.

– **Service Selection:** Once the order has been translated, the RB needs to select the Web services that are required to fulfill the order. The RB will search the UDDI registry if the name of the stock (AMZN) and the stock market (e.g. NYSE) is explicitly provided. If a stock category is provided, the RB will search the DAML-S registries for Web services that deal with the required types of stocks (e.g. *ISP, bookstore,* etc.) in the specified stock market. Consider the following request:

  $\texttt{Req}_6$: { ("inquiry", *, ISP, *, *, *, *) }

  The above order will return a list of all Web services that represent stock markets dealing with stocks in the *ISP* category. It will also return the lowest available price and the number of shares available. Once a Web service is selected by the RB, it registers the service with the *Participants list*.
– **Change Management:** The RB must capture *significant* changes that occur in its registered Web services and react to these changes. It also updates its local *Participants list* when a service joins or leaves the system. *Join* and *Leave* events may be voluntary or may occur as a result of transient or permanent reconfiguration or failure at the underlying communication middleware. When reacting to changes, the RB may decide to abort a pending operation (i.e., submitted but not committed or aborted yet) and re-submit it to another participant that has become more suitable for that particular operation.
– **Response Consolidation:** A request that is broken into several orders and execution threads needs to be consolidated before the response is sent back to the user. For example, a request may require buying a number of shares at the lowest possible price. For these orders, the RB collects the results of all execution threads and selects the thread that produces the lowest price result.

3.2 Request Broker Components

User requests are processed by the Request Broker module. This module: (i) handles communication between the GUI and the underlying layers of the system, (ii) decomposes the requests into execution threads and dispatches each thread to the appropriate Web service, (iii) manages changes in the system, (iv) and consolidates a response for the user. The RB has five components:

– **Communication Interface (CI):** The CI is responsible for secure communication between the GUI and the underlying Web services. To communicate with the RB, the GUI first establishes a secure channel with the CI. The CI is responsible for encrypting and decrypting messages to and from the GUI. The CI also has the task of formatting messages using SOAP standard for communications with the Web services. The CI removes the SOAP envelopes from response messages from the Web services.
– **Request Decomposer (RD):** This component decomposes the request into a list of (sequential and parallel) orders. For example, if the user sends

a request to search for the lowest price for a certain stock, the RD will decompose the request into atomic orders for every stock market currently in the system. In this process, the RD also determines the participant Web services that will be used to fulfill this request. If the RD does not find any participant for the order (e.g., no relevant Web service is currently available), it informs the *Request Manager* which, in turn, aborts the execution of the order and notifies the user.

– **Request Manager (RM):** The RM is in charge of receiving requests from the Communication Interface and sending them to the Request Decomposer. It receives atomic orders and a list of recommended Web services determined by the Request Decomposer. These orders are then dispatched via the Communication Interface to the appropriate Web service. The RM monitors the execution of all orders sent to the Web services. When the execution of *all* the orders terminates and the respective results are collected from the participating Web services, the RM generates an aggregate result and verifies if the service request was *safe*.
  A service request's commitment is safe if and only if:
  • *None* of its orders have been interrupted as a result of the reception by the RM of a notification message.
  • All of its orders have been committed.
  If the order is safe, the RM consolidated the response and sends it to the GUI via the Communication Interface. If the RM cannot commit an order within a system-set time, the RM aborts *all* the execution threads of the order. The RM will also not commit an order if it receives an *abort* message from one or more participating Web services. It then aborts *all* the relevant threads executing at the participants and sends a notification to the Request Decomposer to select another service(s).
– **Notifying Agents (NA):** Notifying agents are processes specialized in monitoring changes in a distributed environment and propagating these changes to entities that need to be notified of these changes. Their goal is to detect, propagate, and initiate reactive processes to changes. They respond to both internal and external change. For all pending orders (i.e., not committed yet), the Request Manager creates one Notifying Agent. This agents periodically send queries to the associated Web service and if there is a change, it notifies the Request Manager.

4 Implementation

The architecture described in the previous section is implemented within a pilot stock market application. The implementation uses state-of-the-art database technologies including Oracle, Informix, and DB2. It also uses Web service technologies, RMI, and database API (JDBC). Mobile agents (implemented using IBM Aglets) are used to detect changes in the system. Web services are described using WSDL and DAML-S. Example of Web services include, NYSE, CAC, FTSE, and TSE services. Each service accesses a backend database to provide the request service.

We generate WSDL descriptions using *Axis's Java2WSDL* utility provided in the *IBM Web Services Toolkit*. These descriptions are published in a UDDI. We implement the UDDI with *Systinet's WASP UDDI Standard 3.1*. *Cloudscape 4.0* database is used to create the registry for the UDDI. Communication between Web services are encapsulated in SOAP envelopes. *Apache SOAP* provides the tools necessary for deploying SOAP messaging.

To simulate the price fluctuation in stock markets, we use a *Stock Market Simulator*. This simulator is a multi-thread Java application. It has four threads, each thread invokes an operation from different types of Web services. The operation updates the *Bid* and *Ask* prices of the stocks in a Web service database every ten seconds. At each update time, the stocks and their update price are randomly selected. Changes in stock prices do not exceed 5%. To make the simulation more realistic, Java threads are used to synchronize the price changes of stocks.

For external changes, we are implementing a simulator function that will randomly change Web service descriptions. Specifically, it changes the status of operation availability in our local UDDI. By removing the description of a service operation, we indicate that the operation is no longer available. Changes are also randomly initiated in the DAML-S registry. We change the category (property) of the Web service to indicate a change in service domain.

The Request Broker is the integral part of our application. It accesses the UDDI and DAML-S registries to select appropriate services. Once services are discovered, its operations are invoked through *SOAP Binding Stub*. The stub is implemented with Apache SOAP API. All messages in the applications are monitored by the agents to determine change.

Users access the system through a Graphical User Interface (GUI). The system's GUI was developed using Java 2/Swing. It consists of two panels. The left panel displays the requests input by the user. The right panel displays all the information returned by the system (e.g., request execution results, registration and authorization information). Users may access the system from any Internet host. All information transfers between the GUI and the Communication Module of the Request Broker occur using a secure TCP connection.

5 Related Work

Web services are slated to be an active research area. We overview some of the research on Web services that is closely related to our work. WebBIS proposes composition and change management for services on the Web [6]. It focuses on the issues of detection, propagation, and reaction to change in service communities. WebBIS uses *ECA rules* and *change operations* to enable change management. WebBIS, however, lacks the support of change management for within the Web service standards. XLANG implements exception handling and transaction rollback by initiating a *compensation* process [17]. This compensation process attempts to "undo" the effects of an incomplete business transaction. This concepts relates to our approach of reacting to change. XLANG does not support our approach of detection and propagation of changes. It only handles reaction through compensation. eFlow uses the notion of *process template* to model composite services [18]. Composers need to browse the *process library* to search for process templates of interest. Furthermore, they need to manually handle interactions and change management between component services when defining composite services.

Commercial platforms are increasingly targeting Web services [19]. *Microsoft*'s *.NET* enables service composition through *Biztalk Orchestration* tools which use XLANG [20]. *IBM*'s *WebSphere* supports key Web service standards [21]. *IONA*'s *Orbix E2A* includes the *Orbix E2A Web Services Integration Platform* [22]. It provides a set of tools for business integration using Web service standards. Developers create Web services from existing applications, including EJBs and CORBA objects. *Sun ONE* (*Sun Open Net Environment*) is a platform for Web services developed by *Sun* [23]. *Sun* began its Web services efforts only recently. Most of these commercial platforms deal with service composition. Changes after composition need to be managed manually. To the best of our knowledge, they provide little or no support for dynamically dealing with changes to services.

6 Conclusion

We proposed in this paper an architecture that supports change management in Web service environments. We describe a generic algorithm for change management in Web service environments. The proposed architecture is based on two key ideas: (i) using ontologies to organize and efficiently select Web services, and (ii) using agents as a mechanism to manage change within the information space.

Acknowledgment. This research is supported by the National Science Foundation under grant 9983249-EIA and a grant from the Commonwealth Information Security Center (CISC).

References

1. Tsur, S., Abiteboul, S., Agrawal, R., Dayal, U., Klein, J., Weikum, G.: Are Web Services the Next Revolution in e-Commerce? (Panel). In: VLDB Conf., Rome, Italy (2001) 614–617
2. Medjahed, B., Benatallah, B., Bouguettaya, A., Ngu, A., Elmagarmid, A.: Business-to-Business Interactions: Issues and Enabling Technologies. The VLDB Journal **12** (2003) 59–85
3. Gottschalk, K., Graham, S., Kreger, H., Snell, J.: Introduction to the Web Services Architecture. IBM Systems Journal **41** (2002)
4. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S.: Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing **6** (2002)

5. Feldman, S.: Electronic Marketplaces. IEEE Internet Computing (2000)
6. Benatallah, B., Medjahed, B., Bouguettaya, A., Elmagarmid, A., Beard, J.: Composing and Maintaining Web-based Virtual Enterprises. First VLDB Workshop on Technologies for E-Services (2000)
7. Papazoglou, M., Aiello, M., Pistore, M., Yang, J.: Planning for Requests against Web Services. IEEE Data Engineering Bulletin **25** (2002)
8. McIlraith, S.A., Martin, D.L.: Bringing Semantics to Web Services. IEEE Intelligent Systems (2003)
9. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic Web Services. IEEE Intelligent Systems (2001)
10. Medjahed, B., Bouguettaya, A., Elmagarmid, A.K.: Composing Web Services on the Semantic Web. VLDB Journal **to appear** (2003)
11. Ouzzani, M., Benatallah, B., Bouguettaya, A.: Ontological Approach for Information Discovery in Internet Databases. Distributed and Parallel Databases **8** (2000)
12. Ankolekar, A., Burstein, M., Hobbs, J.R., Lassila, O., Martin, D., McDermott, D., McIlraith, S.A., Narayanan, S., Paolucci, M., Payne, T., Sycara, K.: DAML-S: Semantic Markup for Web Services, (http://www.daml.org/services/pub-archive.html)
13. Payne, T.R., Paolucci, M., Sycara, K.: Advertising and Matching DAML-S Service Descriptions. In: International Semantic Web Working Symposium, California, USA (2001)
14. Deolasee, P., Katkar, A., Panchbudhe, A., Ramamritham, K., Shenoy, P.: Adaptive Push-Pull: Disseminating Dynamic Web Data. IEEE Transactions on Computers **51** (2002)
15. Maes, P., Guttman, R.H., Moukas, A.G.: Agents that Buy and Sell. Communications of the ACM **42** (1999) 81–91
16. McCanne, S.R.S.: A model, analysis, and protocol framework for soft state-based communication. Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication (1999)
17. Thatte, S.: XLANG, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/. (2001)
18. Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., Shan, M.C.: Adaptive and Dynamic Service Composition in eFlow. In: CAiSE Conf., Stockholm, Sweden (2000) 13–31
19. Vaughan-Nichols, S.J.: Web Services: Beyond the Hype. IEEE Computer **35** (2002) 18–21
20. Microsoft: .NET, http://www.microsoft.com/net. (2002)
21. IBM: WebSphere, http://www-3.ibm.com/software/info1/websphere. (2003)
22. IONA: Orbix E2A, http://www.iona.com. (2003)
23. Sun: Sun ONE, http://wwws.sun.com. (2003)