Webmages: An agent platform based on web services


School of Electrical and Computer Engineering, National Technical University of Athens, 15773 Zographou, Athens, Greece

Available online 10 January 2006

Abstract

WebMages (Web-based Mobile AGEnt System) is a mobile agent platform based on the integration of the mobile agent computing paradigm with Web Services. The resulting platform provides a lightweight agent run-time environment inside a web container, thus adding agent functionality to existing web servers. The platform’s components are deployed as Web Services, with SOAP (Simple Object Access Protocol) over HTTP acting as the necessary communication channel. Potential performance enhancements related to the XML parsing of the transferred data and to the difference in performance between the Apache SOAP library and the Apache AXIS engine are also presented and discussed.

© 2005 Elsevier B.V. All rights reserved.

Keywords: Mobile agents; Web services; SOAP; XML

1. Introduction

During the last years, Web Services have become one of the most important architectures involving cooperation of heterogeneous computer systems. Although computer networks were able to transfer data between different hosts, it was the appearance of Web Services that enabled remote hosts to offer services in a more flexible way. Agent-based computing on the other hand is older than Web Services, and it has been used successfully for the implementation of distributed applications. Agents implement the concept of mobile code, and, in coordination with their flexible architecture, can easily fit into highly dynamic and heterogeneous environments such as the web.

Current solutions for integrating mobile agents in the web infrastructure propose either web servers with build-in mobile agent support [1] or web servers unaware of agents [2]. In the latter, the agents interact as local clients to the web server. Some other solutions have been proposed, most of them based on Java Servlets and the JavaBeans specifications [3,4], while other propose agents written in mark-up languages based on XML [5]. Application scenarios for Web integrated mobile agent platforms have been described in [6–8].

In contrast with existing implementations of web integrated platforms, our approach is based on the Web Services paradigm, where access to the platform’s components is achieved through standard XML messaging. In this way mobile agent support can be added on top of existing web infrastructure, with no need of replacing existing components or installing client software.

The current implementation of the platform can be easily installed on web containers offering Web Service functionality, such as Apache Tomcat [9] with Apache SOAP [10] or Apache AXIS [11] as the underlying SOAP engine. The platform offers a management tool implemented as a set of dynamic pages (Java Server Pages) and a full Mobile Agent API that supports both mobile and stationary agents, agent migration and agent communication.

During the platform’s design, the main goals were flexibility and extensibility. To achieve this, an additional layer that encapsulates communication mechanisms and offers abstract interfaces for communication was introduced. This facilitates the replacement of the communication channel (HTTP) with others like RMI or plain sockets, without affecting the core of the platform.
The platform could be used in a variety of applications. Common applications developed in other agent platforms such as distributed computational codes can be transferred to the WebMages platform. Agents can be used in order to monitor remote web servers, while web crawlers can be developed as mobile agents.

The following section elaborates the requirements, which such a platform must satisfy. Section 3 discusses the architecture of the platform, while results from comparison with other agent platforms are discussed in Section 5.

2. Agent infrastructure requirements

Mobile agents are software entities, which can autonomously migrate from one network host to another during their execution, while they can communicate with each other in order to cooperate and achieve a more complicated task. These needs can be satisfied by a number of architectures and programming languages. Two standards exist for mobile agent platform architectures: MASIF (Mobile Agent System Interoperability Facility) [12] and FIPA (Foundation of Intelligent and Physical Agents) [13]. Agent platforms are implemented in a variety of programming languages, but Java is the most commonly used.

Agents consist of code and state. Code is the logic of the agent, and state is the footprint of the agent in the computer’s memory. When an agent migrates to a remote location, both code and state need to be transferred. While code can easily migrate, state migration follows two different patterns [14]. If state includes the execution stack, the program counter and all the data, the agent can continue from exactly the point he stopped before migrating. This type of migration is referred to as strong migration. If only the data migrates, the migration is called weak migration. The Java Virtual Machine (JVM) offers mechanisms for weak migration through its object serialization/deserialization and dynamic class loading capabilities. Apache Tomcat is one of the most popular Web containers. Apache SOAP and Apache AXIS are also developed by the Apache Foundation, and both offer a flexible and efficient API for the creation of Web Services.

The platform is separated in two major parts, the server and the client side, and will be further analyzed in the following paragraphs. The server-side components are deployed in the web container, while the client side consists only of the user’s browser. The usage of HTTP and SOAP enables the platform components to communicate even if they are behind firewalls.

---

**Fig. 1.** WebMages agent platform components and sample use case.
3.1. The server tier

The current architecture (see Fig. 2) adopts many concepts of the MASIF specification, namely the region, the agency and places that reside on the server side. Opposed to existing agent platforms, the agency and the region expose their functionality as web services. Every web container can host one agency and one region. Only one region is required for each agent system.

The agency web service is the actual agent hosting environment. It is responsible for agent lifecycle management, while it also acts as a communication gateway between agents. The agency is organized hierarchically into places, which help organize groups of agents. Places offer the base for applying agent-specific restrictions, so that they can be administered easier.

The region web service acts as a directory service for the platform. It keeps tracks of important information concerning each agent’s and agency’s lifecycle. To take advantage of the region’s functionality, the agencies need to register to it. They are responsible for notifying the region about events such as the creation of a new place or the removal of an existing place, agent creation, and changes at the agent’s status, migration of agents, etc. Queries to the region can be made anytime by using filters so that the necessary information is discovered. The agents can also request such information, but the actual request is dispatched through the agency.

3.2. The Mediator tier

The above two components are useless without a mechanism that enables their communication. While HTTP/ SOAP is the choice of the communication channel, the direct integration of SOAP calls within the agency would make the platform less flexible and more complicated. The adoption of a new communication channel would require to recompile the whole platform and possibly all the implemented agents. To bypass this problem, a new middle-tier was introduced. This tier, called the Mediator API, is a collection of small classes called Mediators. Each of these classes implements a predefined interface (interface IWorker), or extends an abstract class (Worker). Each Mediator is responsible for a specific communication task. These tasks can be either agency-to-agency or agency-to-region (and vice-versa) communication. This separation is reflected by the existence of two different packages, one for requests concerning other agencies and one for the region. The agency – or any other application using the API – provides the Mediators with the parameters required by every task. By using these parameters, Mediators form the XML messages that represent the SOAP requests. Then, they transmit these requests over the communication channel. When the result is sent back to them via a SOAP response, the mediators store it and wait for the application to collect it.

The main reason for Mediators’ existence is to make the communication transparent to the other components of the platform. Different set of Mediators can be used in order to use alternative communication protocols (such as RMI, XML-RPC, CORBA, etc.), or different sets can coexist in order to provide the platform with multiple communication channels, thus enabling usage of different systems.

3.3. The client side

The second constituting module of the platform is the client side. It is the part of the application that enables user interaction with the platform’s entities. By using his browser, the user connects to a set of dynamic pages (JSP) that allow him to monitor and control the region, the agencies and the agents. The management tool makes possible for the user to create, remove, view information about agents, create or remove places, register agencies to the region, etc. These functions are embedded inside the pages and use the Mediator API in order to communicate with the remote components. Some screenshots of the client interface are shown in Fig. 3.

3.3.1. Agent upload and execution

Agent-based application consist of at least one class extending either the StationaryAgent or the MobileAgent class. This class is packed into a JAR archive among with all the other classes that comprise the agent or the agent may use and are not members of the standard Java API. The file is uploaded to the agency via the management tools by using an HTML form. The management tools use the appropriate Mediator in order to contact the agency and invoke the createAgent() service. When requested, the agency reads the agent’s byte code from the JAR file using a custom class loader, loads the agent’s class to the memory, and instantiates a new agent using the Java Reflection API. The agent is placed inside a wrapper that acts as the actual hosting environment and keeps all the important information concerning the agent. This information consists of the agent’s current state, class loader and unique id, references to the agency, to the ThreadGroup

Fig. 2. Agent platform components.
responsible for the agent’s lifecycle, and information such as the address of the current running location and the home location of the agent. Finally, a new ThreadGroup is created, and the lifecycle of the agent begins.

3.4. Communication mechanisms

Agent communication mechanism is a very important part of the agent platform. There are three different communication mechanisms that enable agents to communicate and interact with each other are implemented in the WebMages platform. The first one is synchronous calls and is used for direct calls to remote agents. The agent’s program flow continues only after the call returns. Similar to the previous scheme is the asynchronous call. The call is formed in the same way as the synchronous calls, but the agent resumes his execution directly after he makes the call. At any given time, he can check if the call has returned its result. The last type of call is the asynchronous call. The call is formed in the same way as the synchronous calls, but the agent resumes his execution directly after he makes the call. At any given time, he can check if the call has returned its result. The last type of call is the asynchronous call. The call is formed in the same way as the synchronous calls, but the agent resumes his execution directly after he makes the call. At any given time, he can check if the call has returned its result.

From the programmer’s view, the three types of calls are performed in a similar manner. The programmer must create an instance of the class SynchronousCall, AsynchronousCall or CallBackCall, depending on the type of the call he wants to make. The object created contains all the required information such as call parameters and the recipient’s address. The agent passes the object to the agent’s makeCall() method, and the call is forwarded to the agent’s hosting agency. The agency uses a mediator to contact the agency containing the call’s recipient agent, and transmits the call over the wire. Whenever a result is returned, the agency either forwards the result to the agent or notifies him, depending on the call’s type.

3.5. Mobility

Agents are divided in two categories. The first one contains agents able to migrate from one agency to another upon user request or when dictated by the agent’s internal logic; while the second consists of agents that execute inside the agency they were created. Agents of the second category simply extend the class StationaryAgent. Agents of the first category extend the class MobileAgent. This class extends the abstract class Agent and implements the IMobileAgent interface. IMobileAgent defines the necessary methods for agent migration.

When an agent wants to migrate, he notifies the agency about his intentions through his wrapper. This happens by simply calling his move() method, providing the destina-
tion address. The agency initializes a negotiation with the remote agency (the agency that the agent wants to move to). Using a Mediator, the local agency invokes the requestRetrieve() method of the remote agency, thus requesting from the remote agency to retrieve the agent. Then, the remote agency, also by using a mediator, invokes the retrieveAgent() method of the local agency. The local agency calls beforeMove(), serializes the agent, thus storing his memory footprint into a string, and returns the agent’s memory footprint and JAR file to the remote agency. Then, it removes any existing references and the wrapper of the agent. Finally, the remote agency reads the agent’s byte code from the JAR file, loads the agent’s class and deserializes the agent, re-creating this way the agent by using the Java Reflection API. The method afterMove() is invoked, the agent’s lifecycle resumes, and the agent continues his execution. We have to note that after the agent migration takes place, the agent does not resume his execution from the point that stopped before moving. This happens because serialization behaves as a form of weak migration.

4. Deployment

As it has been previously mentioned, the realization of our Web integrated agent platform relies on the open source Apache SOAP Project and the Tomcat web server and servlet engine.

According to the Apache SOAP architecture a Web Service can register itself to SOAP with an XML file called Deployment Descriptor. The deployment descriptor is an XML file that tells Apache SOAP everything it needs to properly dispatch an incoming SOAP request to the Java class that is responsible for handling it, in our case the Agency and Region classes. Elements such as the Uniform Resource Name (URN) of the service, the methods exported parameters and encoding rules are described. The realization of the agency and region services requires the deployment of two descriptor files. In Fig. 4, the descriptor file of the agency service is shown.

A major part of this XML file deals with the type mappings of several classes that the agency service handles. Since method parameters and return values must be marshaled between client and server, the SOAP implementation must know how to serialize them. Apache SOAP uses type mappings to determine how Java data types should be marshaled to/unmarshaled from XML so that they can be transmitted on/received from the wire. The type mappings are stored in a type mapping registry, with “org.apache.soap.encoding.SOAPMappingRegistry” being the default name.

Each type mapping carries several pieces of information: a Uniform Resource Identifier (URI) describing the encoding style, a qualified name (qname) for the XML element, the Java class to be encoded from/decoded to, the name of the Java class to act as the serializer, and the name of the Java class to act as the deserializer. The Java classes acting as serializers/deserializers must implement “org.apache.soap.util.xml.Serializer” and “org.apache.soap.util.xml.Deserializer”, respectively.

Apache SOAP though, comes with a Java Bean Serializer/Deserializer that can be used to serialize and deserialize JavaBeans using the SOAP encoding style. By applying this transformation, the public properties of the bean become named elements in the XML form. This is why the region and agency associated classes were implemented as JavaBeans.

The above mentioned makes clear that the deployment of the agent platform does not require installation of new software. This is one of the major advantages of the Web-Mages platform. It “sits” on top of an existing web server, without modifying its content or configuration, even without need to restart the server. Existing infrastructure can be used to deploy large-scale agent applications without any additional cost.

5. Performance issues

As stated before, agents in this platform are divided in two main categories: the mobile and the stationary agents.
While the latter of the two remains at the agency that it was created until the end of its execution, the first one has the ability to transport itself (or ordered to be transferred) to a remote agency. This transportation is being performed by the two agencies: the agency the agent currently resides in and the agency the transportation targets. The two agencies negotiate if they are able to make the transfer, and then the actual transfer is made. Each agent (like every program) consists of two parts: the actual instruction to the machine (i.e., the assembly instructions or the Java bytecodes), and the current “thumbprint” of the program in the machine’s memory. Both these parts are transferred to the target agency, and are assembled together in order to re-form the agent. XML though has not addressed embedding binary data within an XML document. As a sequence the serialized data (the agent, its wrapper, etc.) which include ASCII characters not valid in XML will be part of an invalid SOAP message. To avoid this problem, we used Base64 encoding, which transforms the serialized data to a string containing only the characters ‘A’–‘Z’, ‘0’–‘9’, ‘+’, ‘/’, ‘=’, line feed and carriage return. These characters are valid in XML. While this enables us to bypass the problem of invalid characters inside the XML, it adds additional overhead. Each byte array that needs to be transferred (for example the agent’s bytecode or the serialized agent) increments in size by approximately 30% (Fig. 5), thus requiring more time to transfer over the network, and slowing down the migration. Alternatives to Base64 encoding or compression methods may be used in order to solve this problem.

The first version of the WebMages platform was using the Apache SOAP library as the underlying SOAP engine, while the latest version of the platform conforms to AXIS (Apache Extensible Interaction System), the next generation SOAP project from ASF (Apache Software Foundation). In particular, only the Mediators API had to be modified, because as it has been previously mentioned, the Mediator objects hide the SOAP-RPC functionality from the rest of the platform’s components. AXIS not only features a completely new architecture, but also uses internally a SAX parser instead of the DOM parser utilized in the Apache SOAP library. The main difference of SAX and DOM parsers is that SAX parses large documents without having to store all of the data in memory, while in DOM a parser generally must build the entire tree for the document in memory. This difference reflects in a significant performance improvement when using SAX, especially for large XML documents. The adoption of AXIS led to improvement of the agent’s migration time. Fig. 6 shows the total migration time for different agent sizes. Larger agents result to larger XML documents, and longer parsing periods. Therefore, as the agent size grows, the time saved using AXIS increases.

Fig. 6 shows the time taken for migrating an agent between applications when the size of its state varies between 0 and 3072 Kbyte. The graph also shows corresponding times for directly sending an object of similar size over a bare socket on a Java object stream. The measurements were made using two equal machines (Celeron 1 GHz, 512 Mbyte RAM) interconnected by a FastEthernet switch (100 Mbps). Each experiment was executed 60 times and the results were averaged, resulting in a negligible standard deviation. Memory cache was active, so after the first migration there was no transportation of code. As can be seen, migrating an agent with no state takes about 27 ms. Migrating an agent of 3 Mbyte takes approximately 347 ms. One important observation is that the overhead of migrating large agents is much less than that of migrating small agents. This is easy to explain. Since the migration protocol is basically a request-reply protocol, small agents suffer a higher overhead because of the latency of the network. As for larger agents the transmission of the state takes longer, the initial negotiation cost is hidden. An easy optimization that could largely benefit small agents would be to check the size of the code in advance. If it was small, then the initial negotiation could be bypassed and the code would be sent independently of knowing if the receiver already had it or not. This would improve performance as long as the time spent in sending the code would be less than the time spent due to the latency of the network during the request-reply phase. Another interesting optimization would be to have a distributed hashing method that could be used to determine within a reasonable probability if a host already possessed the code of an agent. These opti-

**Fig. 5. Base64 memory footprint size.**

**Fig. 6. AXIS vs SOAP comparison.**
mizations were not implemented since the migration times were already quite lean and the main focus of the work was not on the optimization of migration protocols. For illustrative purposes, the graph of Fig. 6 shows the results of migrating objects of the same sizes of its corresponding agents by using a bare socket. For small agent/object sizes, migrating an agent is much slower than transferring an object. In the worst case, for an empty agent, it is 5 times slower. Migrating a 256 Kbyte agent is 2 times slower. Migrating a 3 Mbyte agent is only 8% slower. Nevertheless, it must be pointed out that simply sending an object through a bare socket is possibly one of the fastest operations that can be performed: it is just a matter of opening the socket, writing the serialized object and closing the socket. Migrating an agent involves deactivating it, connecting to the other application, negotiating the transmission of the agent and its code using a request-reply protocol, sending the agent, and reactivating its execution on the other side. Although it is quite an elaborate process, sending a mobile agent and sending an object are comparable operations in terms of performance.

6. Conclusions and future work

In this paper, we describe a Web integrated mobile agent platform-based on Web Services. The Web Services programming model perfectly matches the needs of a mobile agent platform, without any drawbacks in extensibility and flexibility. By replacing Apache SOAP with Apache AXIS, we achieved significant performance improvement especially for large messages. Our current activities focus on further improving the performance by implementing a dynamic functionality that will decide at run-time which one of a set of different encoding schemes to use, thus improving performance. Alternative sets of Mediators are being developed, in order to further extend the functionality of the platform and to perform a performance comparison between different kinds of communication channels. Currently, Mediators using plain sockets and RMI for communication are being developed. Finally, the platform is extended in order to provide mobile Web Services. The main idea of this approach is that agents deployed in the platform will expose their functionality as Web Services. Agents will replicate their selves and/or migrate to different hosts in order to provide a set of fault-tolerant Web Services independent of the host they are executing.

References


 Ioannis E. Foukarakis was born in Athens, Greece, in 1980. He received his degree from the National Technical University of Athens, Greece in 2003. He is currently working towards his Ph.D. degree at the National Technical University of Athens (NTUA), Greece, in Electrical and Computer Engineering. His research interests focus on middleware and distributed systems such as Mobile Agents and Grid Computing. He is a member of the Technical Chamber of Greece.

 Antonis I. Kostaridis was born in Athens, Greece, in 1976. He received his Dipl.-Ing. degree from the University of Patras, Greece in 1999 and his Ph.D. degree from the National Technical University of Athens (Greece) in 2001. His research interests include distributed system architectures over heterogeneous networks and Grid Computing. He is a member of the Technical Chamber of Greece.

 Christos G. Biniaris was born in Athens, Greece, in 1977. He received his Dipl.-Ing. degree from the University of Patras, Greece in 1999 and his Ph.D. degree from the National Technical University of Athens (Greece) in 2004, all in Electrical and Computer Engineering. His research interests focus on middleware and object-oriented techniques for the solution of the scale electromagnetic problems. He is a member of the Technical Chamber of Greece.
Dimitra I. Kaklamani was born in Athens, Greece, in 1965. She received the Diploma and Ph.D. degrees from the School of Electrical and Computer Engineering National Technical University of Athens (NTUA), in 1987 and 1992, respectively. From 1993–1994, she was a researcher at the Institute of Communications and Computer Systems, NTUA. From January 1994 to September 1996, she was a Consultant at the Hellenic Telecommunications Organization S.A. In April 1995, May 2000 and October 2004 she was elected Lecturer, Assistant Professor and Associate Professor, respectively, at the School of Electrical and Computer Engineering, NTUA. She is the coeditor of Applied Computational Electromagnetics, State of the Art and Future Trends (NATO ASI Series F: Computer and System Sciences, Vol. 171, New York: Springer-Verlag, 2000) and has published over 140 journal and conference papers. Her current research interests focus on the use of object oriented methodologies and middleware technologies for the development of distributed systems, as well as development of visualization and real-time simulation techniques for solving complex large-scale modeling problems of microwave engineering and information transmission systems. She is a member of the Technical Chamber of Greece.

Iakovos S. Venieris was born in Naxos, Greece, in 1965. He received the Dipl.-Ing. degree from the University of Patras, Greece in 1988, and the Ph.D. degree from the National Technical University of Athens (NTUA), Greece, in 1990, all in electrical and computer engineering. In 1994 and 2000 he became an Assistant and Associate Professor respectively in the School of Electrical and Computer Engineering of NTUA. Since 2004, he is a Professor in this School. His research interests include broadband communications, Internet, mobile and intelligent networks, internetworking, signalling, service creation and control, distributed processing, agent technology, and performance evaluation. He is a member of IEEE and the Technical Chamber of Greece.