Agent-Client Interaction in a Web-based E-Commerce System

Minor Gordon[1], Marcin Paprzycki[1], and Violetta Galant[2]

[1] Computer Science Department, Oklahoma State University, Tulsa, OK 74106, USA
{minorg,marcin}@a.cs.okstate.edu
[2] Department of Intelligent Systems, Wroclaw University of Economics, 53-345 Wroclaw, Poland
galant@manager.ae.wroc.pl

Abstract. In order for agent technology to fulfill its promise, it must be integrated with other existing technologies. In this paper we address one of the problems occurring in a real-world agent integration project—interaction between agents and non-agents. The proposed solution is designed to provide non-agents (clients in particular) access to the agent services, without restricting the agents.

1 Introduction

The promise of agent technology remains unfulfilled. While there exist many scenarios in which agents play a key role, few of them have reached beyond the early stages of development. One of the main reasons for this discrepancy is that agent system researchers and developers have created self-contained worlds for agents to inhabit. Agent environments have been incapable of interacting with each other, as well as with other existing tools and technologies. It is our belief that, in order to help agents take-off, we must start implementing agent-based systems that collaborate with currently available technologies. Here we follow the lead of Nwana and Ndumu who, in their highly critical paper [16], suggest that the best path for agent technology lies in implementing realistic agent systems. This approach means, however, that at first agent systems may not live up to their original promises. To integrate agents with existing technologies often means that we cannot take full advantage of agents' potential capabilities.

As our contribution, we have proposed a working model for one of the classic agent scenarios – a personalized travel support system [1]. Though there have been many agent-related projects in this domain, most of them have either been very limited in scope [15, 17], or have never left the planning stages. The framework for our travel support system is based on the e-commerce model of *management* and *delivery* spheres, linked via a communication channel [9]. This link, as well as the functionality within both spheres, is realized by agents as, aside from their other strengths, agents are an excellent way to decompose complex systems into task-oriented modules [14]. In the content management sphere, the travel-related information is indexed into hierarchical federations according to its ontology and geographical extension. In the content delivery sphere

the personal agent collaborates with other agents (e.g. travel expert agent, advertising agent, data integration agent etc.) to deliver to the user personalized information and targeted advertising. It is the personal agent that communicates directly with the user.

In addition to the problems anticipated in [15, 16], there is one which is only encountered in the implementation phase—the problem of agents interacting with non-agents. For an agent-based e-commerce system, agents must be able to communicate with its users. Currently, the majority of interaction between e-commerce systems and humans is realized through web browsers. However, in a travel support system, we must deal with a vide variety of Internet-enabled clients, and these clients must be able to connect to the agents. The aim of this note is to propose a solution to the agent-client communication problem, granting access to all types of clients while retaining the advantages of the agent paradigm.

2 Agent Communication

One of the implicit assumptions that agent researchers and developers make is the inevitability of this technology becoming ubiquitous to the point that agents will be able to freely move between servers, desktops and personal devices. However, in computational practice the situation is far from reaching this stage. One of the primary technological obstacles is that most of agent platforms require the host device to run a platform-specific virtual *agency* to receive incoming agents. Unfortunately, at this time agencies are not widely deployed, preventing agents from traveling freely between systems.

Since agents are restricted in their ability to travel to Internet-enabled devices, we must look into the possibilities of these devices communicating with agents. While the Foundation for Intelligent Physical Agents (FIPA) standards for agent communication allow agents from disparate platforms to communicate with each other in a platform-neutral language (ACL [4]), these protocols and formats are limited to inter-agent communication. Therefore, software that tries to communicate with an agent has to either (1) use FIPA-compliant Message Transport Protocols (MTPs [5]) and the Agent Communication Language [4] – assuming these are supported by the agent platform they try to communicate with, or (2) connect with an agent via a platform-dependent method such as IIOP. Neither of these options is particularly attractive. The MTPs of (1) are based on standard protocols such as HTTP [6] and WAP [8], but are employed in an atypical fashion, i.e. the basic call-response mechanisms (GET, POST) are the same as in standard HTTP or WAP, but additional headers and specially-formatted body content are **required** by the FIPA MTP standards. This means that an HTTP request submitted by a web browser would not be understood by a FIPA-compliant agent supporting the HTTP MTP (the necessary extensions mandated by the FIPA would not be present). In order to speak with an agent over HTTP or WAP, either both communicators must be FIPA-supporting agents, or one must be an agent and the other must pretend it is an agent by speaking the HTTP "dialect" understood by the FIPA agent. Option (2) requires

the deployment of platform-specific libraries in order to connect to the agents, which is equally restricting. Aside from the numerous interoperability issues across clients, this option requires an addition of multiple libraries to communicator, considerably increasing the size of deployment packages. Non-agents, then, may communicate with an agent, but they must do so on the agent's terms.

We observe a number of client-agent systems that cater to agents by forcing users to download proprietary applets or applications supporting communication. Obviously, this approach is very impractical, and in the case of thin clients such as cell phones, is technically unfeasible. In order to design an interface between clients and agents that will circumvent the above-described restrictions, we start by considering the range of clients our system will serve and their respective limitations.

3 Clients

The client base includes WAP-conversant phones and PDAs as well as standard PC-based web browsers, and even non-human entities such as external agents. Obviously, any e-commerce system should offer a proprietary application and/or applet client for taking full advantage of the system, but this cannot be mandatory. Because of the unpredictability of client capabilities, we can only require a minimum subset of these in order to successfully interface with our system. Clients must be able to display some flavor of markup (HTML, XHTML, WML, etc.) and connect to our services, either through an external gateway (as do WAP devices) or directly over the Internet, using standard protocols such as HTTP (in its standard form).

In order to allow all clients equal access to the services we must also place a number of restrictions on the nature of this access. For instance, we cannot assume clients can process XML (in any fashion), access resources on the user's system, modify the parameters of the communicating protocol, execute Java bytecode or even store cookies. These restrictions preclude atypical communications methods like the the FIPA MTPs described above. Any communication mechanism that might be suspicious to a firewall must also be rejected. These restrictions disallow any *writes* to the client, including downloaded Java applets and applications as well as state persistence. We have to consider the client $read-only$, and adapt the implementation accordingly. It should be noted that from the client's perspective the architecture of the server is largely irrelevant, as long as it provides services/responds to requests in a format and protocol the client understands.

4 Agents as Services

One of the better possible solutions to the above-described problems is to treat agents as services, exposing a service interface like any other software module. This approach is embodied by Hewlett-Packard's Bluestone middleware system – the Total-e-Server [11] in particular. The Total-e-Server implements a Universal

Listening Framework [12] for communicating (bidirectionally) with clients, which include the full range of WAP, HTTP and XML-consuming browsers as well as Java Messaging Service (JMS) nodes, FTP clients, e-mail messages via SMTP etc. Client requests for services, arriving in various formats, are received and translated by an appropriate listener (HTTP, WAP, etc.) and forwarded to the load-balanced Universal Business Server (UBS), which negotiates services with the various modules attached to it (including agents); thus the UBS can act as a "front" for the non-conformant service modules such as agents. The response from the selected service module or modules is transformed into a generic format, forwarded back to the listening framework, and delivered to or picked up by the client. Note that the core of the system (the ULF, the UBS and the Load Balancing Broker) remains firmly rooted in powerful, centralized servers, while retaining some of the scalability of modularized services [3].

This approach is very stable, because it does not rely on agents (or any other particular component type) for services. However, this stability comes at a price, because it reduces the components attached to the service broker to mere request handlers. This limits much of the usefulness of agents as it ties them too closely into the architecture; the mobility, autonomy and wide distribution of agents – three of their defining characteristics [14, 16] – are all inhibited by this configuration.

5 Client-Agent Communication – Proposed Solution

Our approach was developed for the implementation of our travel system, and hile certain components resemble the Bluestone architecture (as depicted in [3]), we arrived at this configuration from an entirely different direction. The result is summarized in 1.

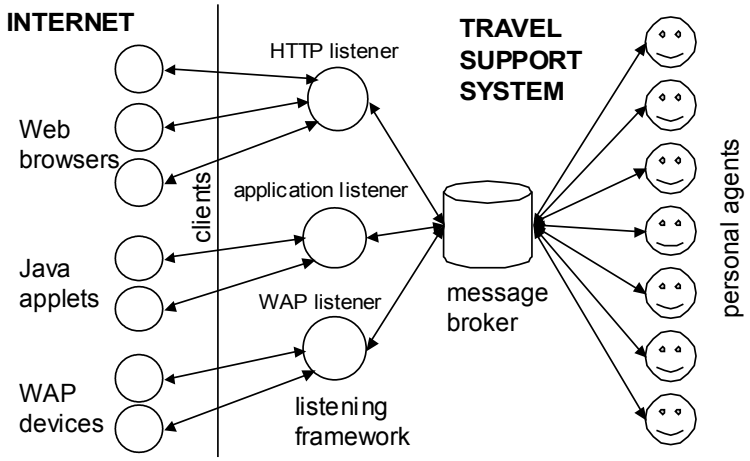

Fig. 1. Proposed solution to the client-agent communication problem.

We start from the ideal of clients connecting directly (one-to-one) to agents. In our system, this connection is instantiated with the personal agent, which is the focal point for all interaction with the client and, by proxy, the user manipulating the client [1]. Given that an actual implementation of a client-agent bi-directional link is currently impossible (see above), we proceed to *emulate* it. This is accomplished by inserting a **limited** number of intermediaries between client and agent. These intermediaries pass requests and responses between the clients and the agents in the system, transforming the communications to appropriate formats as necessary. As indicated in 1, we utilize two intermediaries for performing these functions. This is the minimum; in order for our system to be as extensible as Bluestone or comparable middleware, we would have to insert more intermediaries into the chain. However, the more intermediaries we include, the more we tie the agents to them. As opposed to the strongly-connected core of Bluestone, we are pursuing a loosely-connected conglomeration of components, allowing the agents in the system to be distributed, mobile and autonomous. In this way we conform to the long-term vision of agent research, which would have us remove all intermediaries and send a representative agent to directly interact with the client, using the agents on the server as support (for retrieving and integrating data, inserting advertising, etc.). If/when new agent platforms are deployed or integrated widely enough to allow our agents onto the client's side, our system will be ready for this shift.

Let us now discuss in more detail the components of the proposed solution.

5.1 Listening Framework

As with the Bluestone system, an extensible listening framework is required to communicate with various client types, with one listener per protocol (eg. HTTP, WAP). These listeners transform incoming requests from a protocol-specific format into one that the travel support system can understand, then forward these request-messages to a message broker. Upon receipt of these messages, an agent in the system prepares a response (see [1] for details), wraps it in the same common message format and sends it back to the message broker. Here the response message is picked up by the listening framework, unwrapped, transformed into the protocol-specific format, and delivered.

In the current implementation, only the HTTP listener is active. Requests may come in the form of HTTP GET and POST transactions. These can be handled by CGI programs or server-side scripting languages such as ASP or PHP. This frees us from ties to a specific server platform, such as J2EE (the basis for the Bluestone system [3]) for hosting our listening framework. Any web server with CGI or scripting capabilities will suffice.

Currently, the protocol listeners are not required to maintain the client state – usually done via cookies, which reference session variables on the server – though this may be desirable in an alternate form: enough information can be garnered from the request itself (e.g. the client's IP) to retrieve the client's responses. If future listeners are implemented as statefull, they may still use session variables to store the client's login name, time of last contact with the system, number of

pending requests and other states. In this case a state table will be employed, in lieu of the state-maintaining mechanisms of the server environment.

At this point we encounter the problem of delivering messages to disconnected clients. This inncludes WAP devices as well a standard web browsers, which, without added components (such as applet or Java applications – which we tried to avoid in the first place) cannot listen for callbacks. This is a common problem when dealing with client devices on the Internet. To date there are two possible solutions:

Synchronous request. The client blocks while waiting for a request to be fulfilled by the system. This solution is far from optimal. Client requests may be issued preemptively, in order to present information to the user almost immediately. Blocking the client prevents the user from interacting with it, exploring other options or making more requests.

Interval polling. Clients poll for responses whenever they are ready. A *check responses* request is sent to the appropriate listener, which will then retrieve any prepared responses waiting in the message broker – a mailbox system. More capable clients can spawn threads to poll at set intervals. Thinner clients may be forced to block on intervals to poll; however, the wait would not be long because the listener will either find a prepared response or report that none are ready (no processing involved). Admittedly, this solution is relatively costly on the server side – many clients accessing the listening framework repeatedly – but, given the assumed inability of our system to utilize callback, we deem it necessary. It should also be noted that, in a Bluestone-like framework, our two-intermediary architecture would probably scale poorly, because it lacks the necessary load-balancing components (e.g. Load Balancing Broker [3]). However, as we shall see, our model takes a different approach to scalability.

5.2 Message Broker

Messages – both requests by the client and responses by the agents – are handled by a message broker, which stores them in a common format. For implementation of this functionality we have selected the SOAP standard [18]. This selection is based on our earlier experiments. Our initial implementation employed JMS (Java Messaging Service) as the message broker. While JMS's queues and topics were extremely attractive from the perspective of centralization, stability, ease of creating messages and other factors, there was one major drawback – in order to communicate with the JMS server, both the listeners and any agent receiving messages had to include the JMS support classes. This proved to be unacceptable. Outside the J2EE environment (servlet containers, application servers, etc.), using J2EE components requires a great deal of extra work, and includes much functionality we do not need. Additionally, by using JMS we would tie ourselves to this environment, a circumstance we definitely wished to avoid. For similar reasons we decided to avoid the FIPA ACL, which is a very narrow standard, intended for use by agents, inside of the agent platforms. Its application would require unacceptable (and in some cases, unfeasible) workarounds in order for clients to use it.

SOAP, on the other hand, is platform- and application- neutral. While SOAP messages can be routed by a messaging server or other broker, the headers of the messages allow them to be interpreted and/or routed by the components of the system, without the intervention of a broker program. The primary reason for selecting SOAP as a common message format, however, is that it can be serialized as XML, which means it can be stored as text. Our "message broker" is actually a database, with tables created for each client to send messages to and poll responses from in the way described above.

Here, once again, we encounter our self-imposed restrictions on the assumed client capabilities; they may not be able to read and write SOAP messages. Thus we must allow the client to make a request in format/protocol native to it, and have the listener translate the request into a SOAP message, for storage and subsequent consumption by the agents. Likewise, the responding agent will output the requested data in a format the client will understand (HTML, WML), and wrap it in the body element of a SOAP message. Though SOAP was primarily intended to encapsulate Remote Procedure Calls (RPC) in a platform-neutral format (as opposed to binary protocols such as CORBA, COM/DCOM and Java RMI), SOAP message bodies can also be used to enclose arbitrary literals.

When the client polls for a response, the listener will pick up the SOAP message left by the responding agent(s), unwrap it (processing any headers) and send the body contents to the client. We have two options here: the contents can be encoded directly in the client's "native" format by the responding agent, as described above; or the contents could be output by the agent into schema-conformant XML, then XSLT'd into a format the client can understand by the listener. For simplicity's sake we have chosen the first option for the initial implementation, though the latter may prove to be more extensible, and allow the listeners in the framework to be highly specialized to a wide variety of protocols/formats, beyond HTTP/HTML and WAP/WML. These might include SMTP and SMS, proprietary socket-based protocols, and even the binary object protocols, thus letting the agents be ignorant of any client formats.

There are many added benefits to the message-broker-as-database setup. By storing SOAP messages in a client table, we retain a complete record of exchanges between the client/user and the travel support system. These messages will later be mined by another agent, for use in refining the user's profile, in order to offer more personalized content [1]. On the other hand, one of the few drawbacks to storing SOAP message in a database is that another agent is required to watch and clean up the database, because the SOAP messages in the client tables are never "consumed" (removed) from them. In polling the store, both the listener and the personal agent check for messages newer than the last time they polled, ignoring all previous messages.

5.3 Proposed Solution as an Example of Web Services

By deploying the two-intermediary architecture presented above, the complete travel system can be offered as a web service, and described by the Web Services

Description Language (WSDL). A listener in the framework, instead of interpreting HTTP or another protocol, receives SOAP messages in the standard encoding style for RPCs, and passes these messages into the client's table – with little or no translation required. These SOAP RPC messages are quite similar to the messages generated by the HTTP or other listeners: the calls for agent services are not directly mapped to any particular agent in the system, but are interpreted based upon other parameters in the SOAP header.

One of the most powerful of SOAP header parameters is the *actor* attribute. Any entity (called a SOAP node) which processes a SOAP message may act in the role denoted by the actor attribute (a URI) [10]. This means that the headers of SOAP messages can be specifically directed to nodes (processing entities) playing certain roles. In our model, one of the headers of an outgoing/response message from an agent could be intended for the listener, instructing the listener to perform some translation or add some data (such as pickup time) to the contents of the response message, before it is delivered to the polling client. In the opposite direction, a listener receiving a client request could add headers intended for specific agents in the system, to help them to process the request. Although the personal agent will be the end point for all incoming client messages, other agents may fulfill their roles without interfering with the personal agent's functions (e.g. read-only operations). For example, a logging agent might collect certain headers for debugging or statistical purposes. The advertising agent, which serves relevant advertising to the personal agent from its own database of advertisers, might process the message preemptively and locate relevant ads, so that when the personal agent requests them, they will already be prepared.

5.4 Scalability

In our implementation we attempt to take full advantage of the decentralized nature of software agents. In order to communicate with clients, the agents require the listening framework and message broker intermediaries. However, this does not tie the agents to a specific server or cluster of servers. As long as *a* message database is immediately accessible to both the agents and *a* listener framework, the system is expected to be scalable. Because the message database does not require any global tables (referenced by all parties), it does not need to be unified. Different databases and attached listening frameworks, physically distributed to high use areas (e.g. Los Angeles, New York), would localize the client-server interaction. The personal agent is required to be mobile and follow the user across the country, or even change locations within a large city. Upon notification of the move (initial re-contact), the personal agents condenses all of the messages in the client's table and moves them to the database locale closest to the client's new location. The remaining agents in the content delivery subsystem would not have to transfer, because the majority of them are merely slaves to the personal agent, and need know nothing of the user to satisfy the personal agent's demands.

6 Concluding remarks

In this note we have presented a solution to the agent-client communication problem arising in the Internet-based travel support system. The proposed solution is based on inserting two intermediaries, the listening framework and the message broker and is expected to be scalable. Thus far we have completed the initial stage of implementation, laying out the groundwork for meaningful client-server interaction. In doing so we crossed a number of very interesting bridges:

- The agent platform we chose for the initial implementation of the travel support system was Grasshopper. The main reason for this choice was the extensive documentation which accompanied the platform, the commercial foundations of the company producing it (versus many of the academic projects, which are often spuriously developed). However, in order to take the next step to integrating agents, we will require a platform which more closely conforms to the FIPA standards. This platform remains to be chosen, though JADE [13] and FIPA-OS [2] are prime candidates.
- For our listening framework we tried several approaches. Our first listener was implemented as a simple servlet running a, J2EE-based, Tomcat container. However, considering such factors as size, speed, scalability and access to system resources as essential to the listener framework implementation, our focus shifts to the Apache web server and its associated modules, such as PHP, for basic CGI.
- In selecting a relational database for the message broker component, we surveyed a number of packages, with particular focus on those freely available. Eventually, we selected Postgresql, because it is known to scale well with many simultaneous queries, is actively developed with a large community, and has a number of contributed extensions which are attractive to us (the GIS types in particular). One of Postgresql's more intriguing features is asynchronous notification of database-connected entities. Though these entities must maintain a constant connection to the database, and poll it for notifications, this is much less expensive than periodically executing SELECT queries for new messages, as is customary in such a situation. Unfortunately, as of this writing these features are still in development, and are not fully supported by the Postgresql JDBC driver, which is what we are using to connect the Java-based agents to the message database. For the time being, we are forced to run SELECT polls from both listeners and agents.

In the next stage of implementation we will proceed in two directions. First, we will integrate the client-agent interaction with the remaining components of the system, to produce the first prototype. Second, we will experiment with this same interaction in order to study its performance characteristics and establishs its overhead and potential bottlenecks. We will report on our progress in future notes.

References

1. Angryk, R., Galant, V., Paprzycki, M., Gordon, M: Travel Support System - an Agent-Based Framework. Proceedings of the International Conference on Internet Computing IC'2002, Las Vegas, NV, June, 2002, (to appear)
2. Buckle, P.: FIPA and FIPA-OS Overvie. Invited talk at the joint Holonic Manufacturing Systems and FIPA Workshop, London, September, 2000, FIPA-OS, http://fipa-os.sourceforge.net/. (2000)
3. Burg, B.: Agents in the World of Active Web-services, http://www.hpl.hp.com/org/stl/maas/docs/HPL-2001-295.pdf. (2001)
4. Foundation for Intelligent Physical Agents. FIPA ACL Message Structure Specification, http://www.fipa.org/specs/fipa00061/. (2001)
5. Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Service Specification, http://www.fipa.org/specs/fipa00067/. (2001)
6. Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Protocol for HTTP Specification, http://www.fipa.org/specs/fipa00084/. (2001)
7. Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Protocol for IIOP Specification, http://www.fipa.org/specs/fipa00075/. (2000)
8. Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Protocol for WAP Specification, http://www.fipa.org/specs/fipa00076/. (2000)
9. Galant, V., Jakubczyc, J., Paprzycki, M.: Infrastructure for E-Commerce, Proceedings of the 10th Conference on Extraction of Knowledge from Databases, Karpacz, Poland, May, 2002 (to appear)
10. Gudgin, M., Hadley, M., Moreau, J-J., Nielsen, H. F.: SOAP Version 1.2 Part 1: Messaging Framework. W3C Working Draft 17, December, 2001
11. http://www.bluestone.com/downloads/pdf/06-21-01_Total-e-Server_white_paper.pdf. (2001)
12. http://www.hpmiddleware.com/downloads/pdf/02-27-01_ULFWhitePaper.pdf. (2001)
13. Java Agent DEvelopment Framework (JADE). Telecom Lab Italia, http://jade.cselt.it/ and http://jade.cselt.it/papers.htm (2001)
14. Jennings, N. R.: An Agent-based Approach for Building Complex Software Systems. CACM, 44 (4), (2001) 35–41
15. Ndumu, D., Collins, J., Nwana, H.: Towards Desktop Personal Travel Agents. BT Technological Journal, 16 (3), (1998) 69–78
16. Nwana, H., Ndumu, D.: A Perspective on Software Agents Research. The Knowledge Engineering Review, 14 (2), (1999) 1–18
17. Suarez, J. N., O'Sullivan, D., Brouchoud, H., Cros P.: Personal Travel Market: Real-Life Application of the FIPA Standards. Technical Report, BT, Project AC317 (1999)
18. World Wide Web Consortium, http://www.w3.org/2002/ws/. (2002)