Conversation-enabled Web Services for Agents and e-Business

James E. Hanson, Prabir Nandi, David W. Levine
IBM T.J. Watson Research Center
Yorktown Heights NY 10598
{jehanson | prabir | dwl}@us.ibm.com

Abstract

In this article we outline some enhancements to the existing Web Services architecture and programming model, which will enable them to support the needs of fully-realized dynamic e-business and software agents—which have much in common. Of particular importance is conversation support, *with its core element, conversation policies.*

1 Introduction

The emergence and continued development of Web Services has brought them to the brink of supporting rich e-business applications. The simplified invocation model afforded by SOAP, the standardized, public description of invocation syntax provided by WSDL and UDDI, and the encapsulation of detailed message-transport plumbing behind a standard invocation framework (WSIF) all are essential stepping-stones toward full support of e-business interactions[1]. But at present, Web Services remain a "vending machine" model—that is, they limit themselves to providing a way in which functions can be made available for invocation over the internet.

Rich e-business interactions require a more peer-to-peer, proactive, dynamic, loosely coupled mode of interaction. A fully realized e-business acts as both the "invoker" and "invokee" in two-sided (or multi-sided), multi-step, complex patterns of interaction with other e-businesses. Its internal business processes are under its unilateral control, both as to what to do in any given interaction, and when and how to make changes; while its interactions with other businesses are mediated by public (or at least commonly held) protocols. Even in cases where there is an agreement in place, the business retains control over the extent to which it follows the agreement.

Interacting software agents correspond almost exactly with the above description. In terms of how they interact, differences between agents and fully-realized e-business are largely a matter of scale and emphasis. But from a Web Services architectural standpoint, they are synonymous.

Consider the following two scenarios:

Scenario 1. EB1, an e-business, contacts another e-business, EB2, about purchasing supplies. EB2 replies that the supplies are currently up for auction, names the auction protocol in use, and offers to include EB1 among the bidders. EB1 checks whether it has the business process logic to support that protocol; it does; so it agrees to participate. EB1 participates in the auction. It bids several times, but does not win. It then contacts another e-business, EB3; finds that the goods are available, either at auction or via one-on-one negotiation over price, quantity, and delivery time. EB1, as a result of having lost the previous auction, opts for the negotiation. EB1 and EB3 exchange a series of offers and counteroffers, eventually arriving at a deal. Then they exchange payment and delivery information, confirmation numbers, and so forth.

Scenario 2. Agent1 contacts Agent2 about renting a car. They engage in a dialogue about Agent1's preferences, Agent2's inventory, and so forth. For protocol, they agree to use a modified Agent Communication Language, in which each message contains one of a half-dozen standard performatives to identify the intent of message, and message contents follow a standard car-rental ontology. At some point, Agent1 volunteers the information that it would be willing to pay up to $10 more for a convertible. This uses a non-standard performative. Agent2 cannot process the non-standard performative, so it replies, "not understood". The negotiation continues as if nothing had been said. Some time later, Agent2 asks whether Agent1 wants a child seat. This term is not in the ontology Agent1 said it was using; so it contacts an ontology server to find out about the term, to be told that it relates to "passengers" who are "children". Agent1 looks in its fact store to realize that no information about children is contained in its owner's user profile, and no special overrides have been added for this negotiation. So it replies, "no thank you". The transaction continues and eventually completes to the satisfaction of both parties.

In both of these scenarios, all the decision-making capabilities are well within reach of automation today. Thus, such interactions are possible *if* the Web Services architecture can support them.

At present it cannot. But, as we shall see, this goal is in fact within reach. Many of the shortcomings we will discuss are in fact already recognized, and enhancements are under active development. Others are less well understood.

The interoperability needs of fully-realized dynamic e-business are remarkably similar to those of software agents, to the extent that we can describe a common

architecture for both. The purpose of this article is to explore these common middleware needs of e-businesses and software agents, and to identify the places where Web Services may be enhanced to meet those needs. We believe that with a significant and crucial shift in perspective, coupled with a few technical changes, Web Services can accommodate the needs of both software agents and fully-realized e-businesses.

1.1 Conversational model of interactions

It is useful to describe the model of a business or agent that we are advocating: the *conversational* model of e-business (or agent) interaction. The essential features of

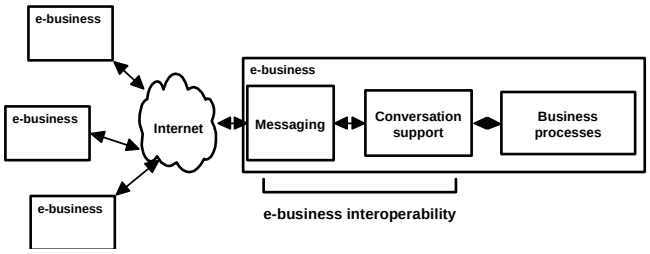

Figure 1

the model are indicated in Figure 1.

A firm's functioning is divided into two broad categories: interoperability technology and business processes. Here, "business process" is a catch-all term for everything that goes on inside an operating firm, such as decision-making, execution of orders, etc., regardless of how or by whom it is done. The interoperability technology is the software the e-business uses to communicate and interact with others, especially other e-businesses.

In the conversational model, the interoperability technology consists of two distinct parts: messaging and conversation support. Messaging is the plumbing needed to send and receive electronic communications with others. Conversation support governs the formatting of messages that are to be sent, the parsing of messages that have been received, and the sequencing constraints on exchanges of multiple, correlated messages. It is a separate subsystem that mediates between the messaging system and the business processes.

The remainder of this article is organized as follows. Sections 2, 3, and 4 take up, in turn, the three parts defined in the conversation model: messaging; conversation support; and business process. Finally, section 5 offers some preliminary work on standards.

2 Messaging

Web Services are often characterized as message-based. This is true; but as commonly illustrated in demoware, it is in fact misleading. In typical use cases and scenarios the Web Service port types define extremely specific input and output formats, which are usually different for each type of functionality exposed by a service. In this section we describe the changes in Web Service usage patterns that will support message exchange of the kind fully-realized e-businesses and agents need.

It is important to note that, in this section at least, we are *not* describing changes in the Web Services technology base; only changes in the way the current technology is used.

2.1 Interaction via message exchange

This means that instead of a client *invoking* functionality exposed as a Web Service, it sends a *request* to the Web Service to have the functionality invoked. Or in other words, the thing that a Web Service exposes is the functionality of receiving a message. Instead of a "getStockQuote" port type, for example, a Web Service would expose a "receiveMessage" port type, to which messages requesting stock quotes are to be sent.

This has the advantage of correctly describing the firm's control boundaries. For example, if a firm exposes a processRFQ service, that implies that it's the *customer* who causes an RFQ to be processed. Really, of course, the firm inserts some sort of control point into the code that gets invoked, whereby the firm makes the decision of whether to *really* process the RFQ (e.g., by calculating a quote and sending it back), or whether to refuse the customer's request. This control point changes the entire meaning of the interaction. It converts the "service invocation" into a "message delivery".

Adopting a message-exchange model from the outset makes the real nature of e-business interactions explicit.

2.2 Generic messaging

This means that delivery of message content is independent of its format. Inputs to port types that can receive generic messages are sufficiently flexible that any content can be delivered in them. In effect, the "receiveMessage" port types should take arbitrary XML documents as input, regardless of schema. The information contained in suggestively-named port types, and in highly-constrained input and output signatures is not lost, however, as we will see.

In generic messaging, arbitrary message content may be exchanged by two interacting parties, even in cases where the recipient of a message is unable to recognize its meaning, make decisions about it, or even, perhaps, parse it. There are two fundamental reasons for this:

Proper assignment of function. Constraining the set of messages that may be sent or received is like programming your telephone to send or receive only a fixed set of words. It is a basic misplacement of function. The messaging infrastructure should not to act as a supervisor defining what may and may not be said. As we

will see below, that job is properly assigned to the conversation support.

Feedback on preferred usage. Unexpected messages may turn out to be valuable, because they may contain clues as to how the they should be handled. The simplest example of this is a message containing a nonstandard abbreviation, which may be guessed at and, by a further exchange of messages, confirmed. Similarly, generic messaging provides a crucial feedback mechanism by showing a business the way in which its customers (e.g.) are attempting to contact it. No business wants to lose a sale because its messaging software refused to deliver the customer's idiosyncratic offer.

2.3 Asynchronous messaging

This means that the output to a "receiveMessage" port type should be little more than a delivery acknowledgment. Return messages, such as the response to a request for information, should be sent via an invocation of a "receiveMessage" Web Service exposed by the original requestor.

This replaces the client-server bias built into the synchronous invoke/return syntax with an inherently peer-to-peer sytle of interaction. Two parties engaged in an e-business transaction use paired, asynchronous messaging port types to send messages to each other.

3 Conversation support

3.1 Long-running conversations

At least as important as the adoption of message exchange is the adoption of "conversation-centric" interactions. This means that messages are sent *within an explicit conversational context*. Messages in conversations are automatically treated as belonging to the same overall, evolving context defined by the conversation itself.

Setup of a conversation is most naturally done via a synchronous request/response pair. The initiator of a conversation invokes a "recieveConversationRequest" port type on the other participant, providing as input information about its own identity and, crucially, a conversation-identifier it has assigned to the conversation. The response can be either refusal or acceptance. In the latter case, the responder must include in the response a conversation-identifier of its own.

Participants then exchange messages asynchronously, but each sender also includes the recipient's conversation-identifier (obtained during the setup) in the message. This permits e-businesses to carry on multiple conversations simultaneously.

Messages received in conversations are, in effect, placed in a conversation-specific inbox created during conversation setup and labeled with the recipient's conversation-identifier. Two e-businesses, in setting up a conversation, each create an inbox for that conversation only, then exchange identifiers for those inbox, to be included in each message the other party sends.

Adopting conversation-centric interaction amounts to recognizing that in the real world of electronic commerce, interactions typically consist of multiple correlated messages.

3.2 Conversation management independent of message delivery

As we said, the messaging subsystem encapsulates the sending and receiving of messages, making it possible to support multiple transport mechanisms (e.g., XML over SOAP, JMS, etc.) by simply plugging them in. This is already part of the extant Web Services standard.

The conversational session, and the logic that manages it, are not dependent on the particular transport used. Messages going in one direction need not use the same transport as messages in the other, and transports may even be renegotiated during the course of the conversation—for example, to increase or decrease the security level, bandwidth, etc.

3.3 Conversation Policies

In machine-to-machine conversations, free-form dialogs are not really practical. Therefore, e-business interactions will make frequent use of preprogrammed interaction patterns called *conversation policies* (CPs). CPs are the heart and soul of conversation support.

Conversation Policies have received much attention in the software agents community[3,4]. However, that work typically blends the notion of conversational state with the notion of the agents' internal processing states—what we are here calling the business processes.

For our purposes, a conversation policy is a machine-readable specification of a pattern of message exchange in a conversation. CPs consist of message *schema*, *sequencing*, and *timing* information. Conversation policies are what take the place of the suggestively-named port types and highly constrained input and output signatures typically seen in Web Services demos.

Figure 2 shows a schematic of a simple CP, in which two participants, A and B, trade bids and counter-bids until one or the other of them accepts the current bid or gives up. Nodes in the graph correspond to different states of the conversational protocol. In effect, each node represents a summary of what has transpired so far in the conversation. Edges connecting nodes correspond to messages being sent by one or the other party, and specify the format or schema of the message as well as which party is the sender. For example, in the starting state (labeled "Start") there is one transition, labeled "A->B: Request Bid", which corresponds to A sending a message to B of the form "Request bid". The CP does not define any other way for the conversation to proceed from its starting state. Similarly, there are two transitions out of the

state labeled “Request Pending”: one in which party B sends a message to party A of the form “Bid = x” (where x represents some value determined by B), and another in which party B sends “Bye”.

In carrying on a conversation, each party separately maintains its own internal record of the conversation’s current state, and uses the CP to update that state whenever it sends or receives a message. For example, at the beginning of a conversation that follows the CP in Fig. 2, A is in the “Start” state of the CP. If and when it sends a “Request Bid” message to B, it changes its current state to the “Request Pending” state. Similarly, B is initially in the “Start” state; if and when it receives a “Request bid” message from A, moves to the “Request pending” state. If B then sends “Bid = x”, it updates its current state to “A’s reply pending”. Or, alternatively, if it sends “Bye”, it updates its current state to “Terminate/Failure”. When A, currently in the “Request pending” state, receives a message from B, it checks to see whether the message is “Bid=x” or “Bye”, and then updates its own current state accordingly. And so forth.

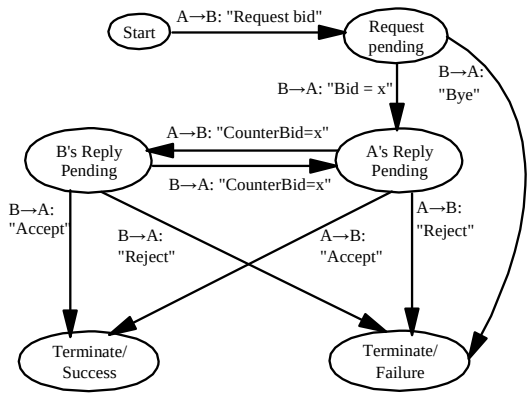

Figure 2

The sender of a message usually (though not always) has to make a *decision* as to which of the possible alternative messages to send, and often supply *data* as well--e.g., the value to fill in for a bid’s amount. Similarly, the recipient usually has to *classify* the message--identifying which of the possible alternatives was sent--and often *parse* it to unpack the data supplied by the sender.

As written, the CP takes a third-person point of view. This permits the same CP to be used by both parties, with the decision of which role to take being made at runtime, possibly as the result of a prior negotiation.

CPs enable extensive reuse of messages. Because a message is interpreted with respect to the conversation’s current state, the same message can be safely reused in multiple contexts. For example, the message “OK” can be used in a bid/counterbid CP to signify acceptance of a bid, in an RFQ CP to signify acceptance of a quote, and so forth. In all cases, the contextual information supplied by the CP and the conversation’s current state removes any ambiguity.

CPs provide economy of expression. The conversational context obviates the need for repeating information already sent, or for including extra information in a message in the mistaken belief that the other party might want it.

Because each of the conversing parties maintains its own record of the conversation’s state, and uses its own CPs to update that record, the parties need not, in fact, be using exactly the same CP. The minimal requirement is that, in the course of a particular conversation, the sequence of messages they exchange corresponds, on each side, to some path through the particular CP that party is using.

3.4 Nested Conversation Policies

In day-to-day business, a firm’s interactions with other firms tend to be made up of common, conventional interaction patterns. That is to say, its conversations tend to have phases or “stanzas” which fall into common patterns, and are reused in different contexts. For example, first there might be discussion of product discovery, then negotiation of the deal, finally settlement. And it is nested: Product discovery, for example, might start with the customer expressing needs, the seller asking pointed questions about them and then recommending a list of possible matches, followed by the buyer making a selection from the list. Negotiation might start with a discussion of the way to negotiate: haggle over price, or place bids in an auction, or etc., followed by, in both cases, a pattern of message exchange appropriate to that negotiation method. After the products are dealt with, then the parties might turn to a dialog about delivery options (if the goods are physical) and prices. Similar, settlement might start with an enquiry into the methods of payment supported, followed a selection of one of them.

Conversation policies are inherently nestable. This means that, as part of carrying on a conversation that obeys a given policy, the conversing parties might choose to start a new conversation policy as a “sub-conversation”, possibly carry it out to completion, then return to the previous conversation policy. In effect, both parties carry on a more narrowly scoped “child” conversation within the enclosing context of the more broadly-scoped “parent” conversation.

For example, two parties might be engaged in a simple negotiated bidding procedure, in which they first identify a set of services to be performed, then they engage in an iterated bidding procedure to settle on a price. That iterated bidding procedure may be represented by the CP in Figure 3, in which the specification of the goods outside the scope of the CP--it is part of the context--and the messages only pertain to the bid price.

In this case, the CP governing the "parent" conversation would contain transitions for starting up a

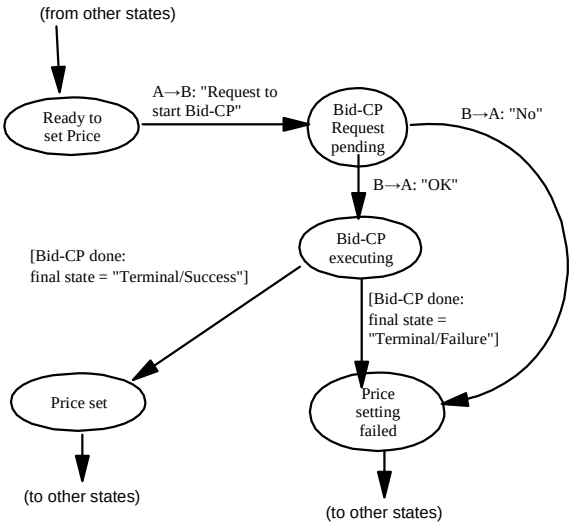

Figure 3

sub-conversation following the bid/counterbid CP. This is shown in Figure 3.

3.5 Pre-/Post-condition CPs

Another style of CP, more familiar to the software agents community, represents an interaction in terms of coarse-grained states connected by pre- and post-conditions. Instead of the fine-grained state structure of Fig. 2, for example, one represents the act of negotiation as in Fig. 4. There, a single state and a single transition represents the exchange of all messages related to the bidding processes. Transitions to the terminate/success

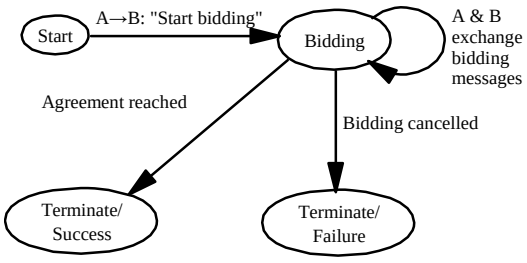

Figure 4

and terminate/failure states are taken when an agreement is reached or when bidding is cancelled.

Pre-/post-condition CPs can be mapped onto fine-grained CPs by explicitly defining separate transitions for all the different bidding messages. But this can lead to an undesirable proliferation of states, for example when the conversation involves multiple attributes being negotiated over simultaneously, with agreement on all attributes required before the post-condition is satisfied. In such a case, a separate fine-grained CP state would be required for each of the possible states of partial agreement—e.g., "X and Y agreed to, but Z and W not."

4 Business process

The conversational model strictly refrains from imposing any particular architecture on the business processes used to make the decisions that drive the conversations. Still, there are two points to be made about business processes in this context.

4.1 Separation from conversation management

This means that conversation support is provided by a subsystem kept separate from the firm's actual business processes. If the business process is managed by a workflow system, for example, the conversation management is *not* part of the workflow. Rather, it is a separate subsystem that is coupled to the workflow as appropriate.

The main reason is that the interoperability technology shouldn't place constraints on how the core of the business works. The business processes are what the interoperability technology is supposed to *support*, not prescribe. They are the thing that differentiates one firm from another; the thing that is most crucial to success and survival; and not the kind of thing a firm would like to expose to the world. Interoperability means connecting up the business processes with the economy--not turning the business over to someone else.

Controlling the business processes is the core of what it means to be an independent business engaged in trade. Each party in a trade, by definition, makes decisions unilaterally and executes them under its own control. Even when under contract, a firm's freedom of action is not compromised, because its decision to obey the contract is unilateral (as, of course, was its decision to sign the contract in the first place). To the extent that "interoperability" comes to encompass a firm's decision-making and/or execution processes, that firm is not engaging in trade--it is obeying directives.

Furthermore, it is futile to try. From outside, there is no way to tell for sure whether a firm is unable to execute a purchase order (for example), or unwilling to do so.

Other considerations:

Maintainability. Business processes change on different timescales from interoperability technology. Changing a business process needs to be done at a firm's instigation, on the firm's own timescale. It should not be dependent on its customers, suppliers, and trading partners. Changes in interoperability technology are, by definition, on a shared timescale.

Ease of modification. Changes in interoperability can be accomplished by something as easy as downloading an XML file for a new CP, then adding new bindings that connect the CP to the business process. Therefore, changes in business processes are neither forced by changes in interoperability technology, nor hindered by it.

Though interoperability technology and business processes are clearly linked, just as clearly they are separate endeavors with separate driving forces, requirements and timetables.

4.2 Proactivity

Unlike the usual Web server architecture, unlike the J2EE programming model, and most particularly unlike the current Web Services programming model, conversational interactions require *proactivity* on the part of the participants.

At the very minimum, it is clear that one of the parties must initiate the conversation; hence one must be proactive rather than purely reactive.

Furthermore, in order to decouple the act of receiving a message from the act of processing it and (possibly) sending a message in response, it is necessary to support a narrowed version of proactivity that might be called *asynchronous reactivity*. That is to say, an incoming message arrives in the recipient's inbox, and a delivery acknowledgement is given as the return code. This completes the activity triggered by the sender. But the message still needs to be processed, data extracted from it and sent back to the business process, and a decision made as to what further action to take, what reply to make, etc. This means the recipient must at least have the equivalent of a timer service or event-dispatch thread running in the background, to pick up where the message-delivery activity left off.

These two examples merely illustrate the minimal degree of proactivity required to carry on a conversation at all. Clearly, within an e-business or software agent, there are many other cases in which a long-running thread is needed, such that the business can carry out its functioning independently of whether anyone is sending it a message or not.

The fact that it is needed both to drive the interactions, and within the business process, indicates that a mechanism of providing for proactive behavior has a place in the Web Services programming model.

5 Evolution of Standards

We close with a preview of two extensions to industry standards that are motivated by the conversational model.

5.1 Conversational extensions to the Java Connector Architecture (JCA)

The JCA [5] architecture provides a set of abstractions for connecting the J2EE platform to heterogeneous Enterprise Information Systems (EISs). The abstractions, defined as a set of contracts at the system and at the application level provide a collection of scalable, secure and transactional mechanisms that enable the integration of EISs with application servers and enterprise applications.

In this work we propose to extend the JCA application and system contracts to support conversations and conversation policies. This will extend the architecture from the realm of EIS integration to cross-enterprise integration. The (conversational) adapters built conforming to the architecture, provides the guarantee of being able to run on any J2EE platform and avail of the system specific resources (transaction, security, connection pooling and *conversation management*).

5.2 Conversation Policy XML

We are currently developing Conversation Policy XML (cpXML), an XML dialect for describing conversation policies. It permits CPs to be downloaded from third parties (such as standards bodies, providers of conversation-management systems, or specialized protocol-development shops). Once downloaded and fed into a firm's conversation-management system, bindings are added to specify the connections between the decision points of the CP and the firm's business logic.

cpXML is intentionally minimalist, restricting itself to describing the message interchanges as we sketched them in Section 3. Thus, for example, it does not cover the way in which the CP is bound to the business logic. It takes a third-party perspective, describing the message exchanges in terms of "roles" which are assumed at runtime by the businesses engaged in a conversation. It supports nesting of conversation policies and time-based transitions (such as timeouts on waiting for an incoming message).

Its first use will perhaps be as a standard of comparison for evaluating forthcoming developments in flow languages such as WSFL, etc. At this time, it is impossible to judge wither a separate language is needed for specifying conversation policies, or whether hybridized flow/state-machine description language will be practical.

6 References

1. For the acronyms in this paragraph, see http://alphaworks.ibm.com/webservices
2. See, for example, M. Greaves and J. M. Bradshaw, editors, *Proceedings of the Autonomous Agents '99 Workshop on Specifying and Implementing Conversation Policies*, (1999)
3. M. Greaves, H. Holmback, and J. Bradshaw, What is a Conversation Policy?, in ref. [2].
4. J2EE Connector Architecture 1.0 Specifications from http://java.sun.com/j2ee/connector/