Transformational Interactions for P2P E-Commerce

Harumi Kuno, Mike Lemon, Alan Karp
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-143 (R.1)
October 10th , 2001*

E-mail: harumi_kuno@hp.com, mike_lemon@computer.org, alan_karp@hp.com

service composition, service conversations, UDDI, WSDL, WSCL, SOAP

Both peer services and web services offer a perspective of services in the role of resources that can be combined to enable new capabilities greater than the sum of the parts. However, current service composition solutions seem to support either highly dynamic discovery or else very loosely coupled service development, but not both. We propose a facilitator service mechanism that can leverage "reflected" XML-based specifications (borrowed from the web service domain) to direct and enable coordinated sequences of message exchanges (conversation) between services. We extend the specification of a message exchange with the ability to specify transformations to be applied to both inbound and outbound documents. We call these extended message exchanges transformational interactions. The facilitator service can use these transformational interactions to allow service developers to decouple internal and external interfaces. This means that services can be developed and treated as pools of methods that can be composed dynamically.

* Internal Accession Date Only Approved for External Publication

To be published in and presented at HICSS-35, Hawaii International Conference on System Services, January 2002

Transformational Interactions for P2P E-Commerce

Harumi Kuno, Mike Lemon, Alan Karp
Hewlett-Packard Laboratories, Palo Alto, CA
harumi_kuno@hp.com, mike_lemon@computer.org, alan_karp@hp.com

Abstract

Both peer services and web services offer a perspective of services in the role of resources that can be combined to enable new capabilities greater than the sum of the parts. However, current service composition solutions seem to support either highly dynamic discovery or else very loosely coupled service development, but not both. We propose a facilitator service *mechanism that can leverage "reflected" XML-based specifications (borrowed from the web service domain) to direct and enable coordinated sequences of message exchanges (conversations) between services. We extend the specification of a message exchange with the ability to specify transformations to be applied to both inbound and outbound documents. We call these extended message exchanges* transformational interactions. *The facilitator service can use these transformational interactions to allow service developers to decouple internal and external interfaces. This means that services can be developed and treated as pools of methods that can be composed dynamically.*

1 Introduction

Peer-to-Peer (P2P) technology brings distributed computing capabilities to individuals, creating a new perspective of network-capable devices in the role of peer service resources that can be combined dynamically to enable new capabilities greater than the sum of the parts. This goal of enabling autonomous services in a dynamic environment to interact in a loosely coupled manner is shared by the web service community. Thus, there are a number of common issues that both peer service and web service developers must address, including:

- *discovery*: how to register and find services in a dynamic environment.
- *interfaces*: how to invoke services once they have been discovered.
- *protocols*: how to coordinate service interactions between extremely loosely-coupled services.

The two communities have distinct approaches, however. Web services exist in a relatively static environment of registered hosts, and static IP addresses, and can leverage a number of standards and protocols (e.g., WSDL, WSCL, WSFL) that support the composition of loosely coupled services. Peer systems support a highly dynamic discovery model, but currently tend to require application-specific centralized control of the distributed program, or else require the download and installation of application-specific software on participating parties, or both.

It follows naturally that the two communities should leverage each other's strengths. Schneider [1] forsees a convergence of peer and web services, where both peer and web services use WSDL and SOAP for service descriptions and invocations. However, even with services using WSDL, SOAP, and WSCL, the question still remains of how to coordinate two services that were independentally developed and may not conform to matching message document types. This problem is exacerbated by the highly dynamic and low-barrier-to-entry characteristics of the peer environment. That is to say, given that P2P discovery mechanisms allow a given peer to discover peers that provide desired resources in a highly dynamic and distributed fashion, we would like a peer service to be able to interact with services provided by other peers without having to download or install new application-specific software.

Previously, in [2], we introduced a lightweight dynamic conversation controller for E-Services. This "third-party" external service can direct and track spontaneous conversations between services and clients, thus enabling services to carry out an entire conversation without the service developers having to implement any explicit conversation control mechanisms. This functionality is especially useful in a P2P environment, because it means that two services can interact without their developers having to implement matching conversation policies. Furthermore, because the controller can be implemented in a stateless manner and because the controller is only responsible for conversation logic (as described in Section 4), conversing parties can switch between any number of instances of a controller in the course of a conversation.

However, our early controller still required service developers to code to matching message document types. This limitation is particularly unacceptable for the P2P environment where it seems more realistic to expect there to be a diverse collection of continually evolving document types than a set of fixed standards.

We introduce *transformational interactions* to help address the problem of how to design and implement systems that are decentralized and conducive to dynamic and autonomous interactions between independently developed applications. We advocate a service-oriented paradigm in which we treat services as pools of functional endpoints that can be composed using typed message exchanges (interactions). A *transformational interaction* is an abstract specification of a typed message exchange that has been extended with the ability to specify transformations to be applied to both inbound and outbound documents.

We propose extending the P2P environment with *facilitator services* that can leverage "reflected" XML-based specifications to direct and enable coordinated sequences of message exchanges (conversations) between peer services. We use the term *facilitate* because the role of such services is to enable, as opposed to control, P2P activity. Separating between conversational logic and interface logic allows us to use conversation and interface specifications to direct services in their interactions, thus freeing service developers from having to explicitly program conversational logic. Extending conversation specifications with transformations allows us to free service developers, to some extent, from the responsibility of navigating message type mismatches. In addition, it allows the services to offload some computational responsibility to the facilitator.

For example, Figure 1 sketches two "shopping cart" conversation specifications for digital photograph storage services (similar to services offered by companies such as ofoto.com). These conversations are specified from the perspective of the photo storage service. The circles represent interactions; the boxes represent inbound document types, and the arcs between the circles represent the transitions between interactions, which are driven by outbound document types. The two conversations are structurally similar, but not identical. The "secure album" conversation requires the client to sign in before selecting an album, whereas the "anonymous guest" conversation doesn't require the client to sign in until they are ready to purchase some photos. In addition, once the client is ready to check out, the "secure album" conversation expects the client to send a document of type "CheckoutRQ" and the server to respond with a document of type "Bill," while the "anonymous guest" conversation expects the client to send a "RequestInvoice" document the service to respond with an "Invoice" document.

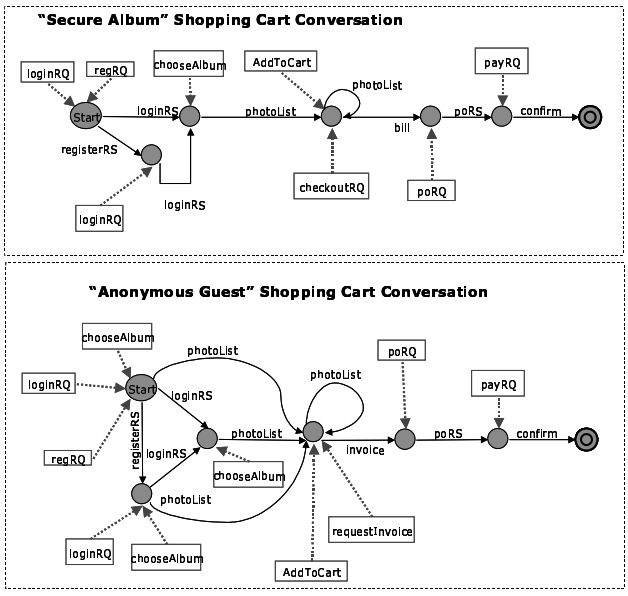

Fig. 1. Two example conversations.

Figure 2 shows how a procurement service developed to support message types from the "Secure Album" conversation can use any instance of a conversation facilitator service to transform messages and direct message exchanges when interacting with a photo service developed to support message types from the "Anonymous Guest" conversation.

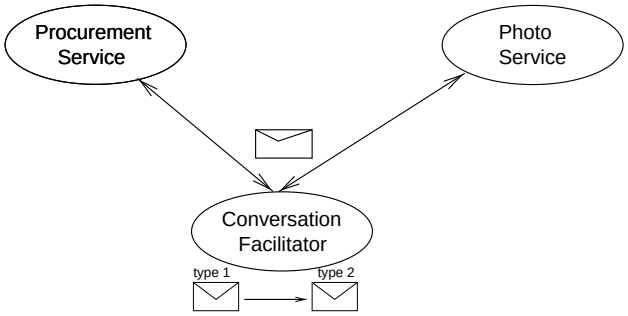

Fig. 2. A conversation facilitator can help two services engage in an interaction.

In this paper, we describe how we leverage web service technologies to create a facilitator service that enables the dynamic coordination and composition of P2P services. For example, our facilitator would enable a peer service that supports only the "anonymous guest" conversation to interact with another peer service that supports only the "secure album" conversation– without having to update either service. In fact, the service developers would be freed from having to implement any conversational logic. We begin by discussing the similarities between B2B and P2P services (in Section 2), and discuss how P2P can benefit from work currently being done in the B2B domain. We then introduce a model and paradigm for facilitated P2P computing. We have implemented a prototype facilitator service, and we describe its architecture in Section 3. Facilitated computing is related to a number of other efforts, which we discuss in Section 5. Finally, we present future work and conclusions in Section 6.

2 Applying a B2B Perspective to P2P

A number of articles have considered how B2B web services can benefit from the P2P model [3], [4]. Here, we take a different perspective, and examine instead what P2P can learn from web services.

2.1 A Web-Service Perspective

Electronic Commerce is driving distributed computing towards a model of service based interactions, where corporate enterprises use web-services to interact with each other dynamically [5]. Web-services are much more loosely coupled than traditional distributed applications. This difference impacts both the requirements and usage models for web-services, and motivates our comparison to the P2P model. Web-services are deployed on behalf of diverse enterprises, and the programmers who implement them are unlikely to collaborate with each other during development, yet the purpose of web-services is to enable business-to-business interactions. Therefore, B2B services must support very flexible, dynamic bindings. Services should be able to discover new services and interact with them dynamically

without requiring programming changes to either service.

For example, a service in one enterprise could spontaneously decide to engage a service fronted by another enterprise. In order for the two services to interact with each other dynamically, they must be able to do three fundamental things. First, clients must be able to discover services. Second, a service must be able to describe its abstract interfaces and protocol bindings so that clients can figure out how to invoke it. Third, the services must be able to carry out complex interactions (conversations) in order to transact their business (for example, the client should be able to login and then purchase an item from the service).

A number of currently developing B2B standards address these issues. Universal Description Discovery and Integration (UDDI) [6] defines how to publish and discover information about web services. The current version of the Web Services Description Language (WSDL 1.0) [7] is a general purpose XML-based language for describing the interfaces and protocol bindings of web service functional endpoints. WSDL also defines the payload that is exchanged using messaging protocols such as SOAP. Neither UDDI nor WSDL currently addresses the problem of how a service can specify the sequences of legal message exchanges (interactions) that it supports. (We use the term "conversation" from the agent community to refer to a legal sequence of message exchanges.) Conversation languages such as HP's Web-Service Conversation Language[1] (WSCL) [8], [9] and RosettaNet's PIPs [10] address this issue, providing XML schemas for defining legal sequences of documents that web-services can exchange. Conversation definition languages and interface definition languages are highly complimentary – the latter specifies how to send messages to a service and former specifies the order in which such messages can be sent. The advantage of keeping the two distinct is that doing so allows us to decouple conversational interfaces from service-specific interfaces. This means that a single conversation specification can be implemented by any number of services, independent of the protocols supported by the various implementations.

2.2 A Model for Facilitated P2P Interactions

Many of the characteristics of B2B services are equally applicable to P2P computing. Both the B2B and P2P models require interaction between loosely coupled decentralized peers. Both environments are subject to difficulties coordinating service developers. We propose to apply the B2B service model of interactions in order to extend peer services with the ability to support very flexible, dynamic bindings.

In our model, peer services (referred to as *services* for the remainder of this document) interact by exchanging messages. Each message can be expressed as a structured document (*e.g.*, using XML) that is an instance of some document type (*e.g.*, expressed using XML Schema). A message may be wrapped (nested) in an encompassing document, which can serve as an envelope that adds contextual (delivery or conversation specific) information (*e.g.*, using SOAP). We define a conversation to be a sequence of message exchanges (interactions) between two or more services.

We define a *conversation specification* (also known as a *conversation policy*) to be a formal description of "legal" message type-based conversations that a service supports. We define an interface specification (also known as a service description) to be a formal description of a service's functional endpoints. We assume that both conversation specifications and service interface descriptions will be published so that service developers can determine how to invoke and converse with a newly discovered service.

2.2.1 Conversation Logic versus Interface Logic

We separate a service's interface logic from the conversation logic that a service might support. This distinction allows us to treat services as pools of interfaces that can be specified by individual participants and then later composed using separate conversation specifications. We can then create conversation controller services that can use conversation and interface specifications to direct services in their interactions, thus freeing service developers from having to explicitly program conversational logic. Such a single third-party conversation controller could leverage "reflected" XML-based specifications to direct the message exchanges of Peer services according to protocols without the service developers having to implement protocol-based flow logic themselves. The conversation controller can assume responsibility for the conversation logic, leaving service developers free to focus on service-specific logic. For example, the controller would handle exceptions due to message type errors, while the service would be responsible for handling exceptions related to message content.

The advantage of this approach is that it enables services to be easily and flexibly composed with a minimum of programming effort. In order to participate in a given conversation type, a service need only to be able to accept and produce messages of the appropriate types. This allows services and clients to discover each other and interact dynamically using published specifications.

That is to say, because the conversation policies are not service-specific, services and clients can interact even if they were not built to use precisely matching conversation policies, as long as both parties are capable of sending and receiving appropriate messages. Furthermore, because the service interfaces and the conversation policies are decoupled, different instances of a service could name their methods independently, e.g., a client could use the same conversation specification to talk to two different book-selling services, despite the fact that one service supports a Login method while the other uses a corresponding Sign-on method.

Finally, this approach gives us a scalable mechanism for handling the versioning (evolution) of conversation policies. Services would not necessarily have to be updated in order to support new or modified conversation policies.

[1] WSCL was originally called the Conversation Definition Language (CDL).

2.2.2 Extending Conversation Definitions with Transformation Specifications

Thus far, we have explained that if peer services communicate by exchanging typed messages, then services can advertise and discover each other using registries such as UDDI, services can publish specifications describing their functional interfaces so that new clients can know how to invoke them, and third-party conversation controllers can use message type driven conversation specifications to direct services in their conversations. When we move this model to an even more decoupled P2P environment with computationally limited peer services, we encounter new issues. For example, the document type mismatch problem (where a service does not support the exact document types required by a conversation specification) becomes even more likely as the number of diverse services increases. Furthermore, peer services may be computationally limited (perhaps running on a small device such as a wristwatch) and may not be capable of performing certain functions (e.g., transforming a document or calculating the difference between two meeting times).

We therefore extend the conversation definition language by allowing the specification of document transformations (expressed using some sort of transformation template, such as XSLT) that can be applied to incoming and/or outgoing documents. We also extend the conversation controller mechanism with the ability to interpret the extended conversation specifications and apply the appropriate transformations.

With this approach, we can enable peer services that have been developed independently to engage in flexible and autonomous, yet potentially quite complex, interactions (conversations) that can exceed the functionality of the individual services. For example, in the example from Figure 1, two services may not be independently capable of engaging in business transaction with each other because neither service has any means of negotiating the document type mismatches. Extending a conversation specification with document transformation templates allows us to delegate the problem of document transformations to the facilitator service.

3 Paradigm for Facilitated P2P Interactions

Our facilitator service can act as a proxy to a peer service and track the state of an ongoing conversation, based on the types of messages exchanged. In addition, as described above, the facilitator can also apply optional transformations to both incoming and outgoing messages at each stage of a conversation. A facilitator service that acts as a proxy performs the following tasks:

- Once it has received a message on behalf of a peer service, the Facilitator service can dispatch the message to the appropriate service entry point, based on the state of the conversation and the document's type.
- When forwarding the response from the peer service to the client, the Facilitator service includes a prompt indicating valid document types that are accepted by the next stage of the conversation. This prompt can optionally be filtered through a transformation appropriate to the client's type. (E.g., if the client is a web browser and has indicated that it would like form output, then the Facilitator service may transform the response into an HTML form before sending it to the client.)
- The Facilitator service can use document transformations specified by the conversation to correct for document type mismatches and to manipulate the document type based on message content.
- If the client requests it and specifies appropriate entry points, the Facilitator service can also direct the client's side of the conversation. This means that neither the service nor the client developer must explicitly handle conversational logic.

In order for a peer service to use a Facilitator service as a proxy, the service developer must do the following (note that the service developer does not need to implement code to handle the conversation flow logic):

- Document the type-based inbound document handling entry points in a specification (ideally capturing both input and output document types).
- Advertise the service with an entry point going through the Facilitator service.

Each time the Facilitator service receives a message on behalf of the service, it will identify the current stage of the conversation and verify that the message's document type is appropriate; if not, then it will raise an exception. If the message is of a valid type, then the Facilitator service will invoke the service appropriately. If not, then it will attempt to apply appropriate transformations to convert the message into a valid type. (If it is not possible to convert the message into a valid type, then the facilitator will raise an exception.) Once the message has been appropriately converted, then the facilitator will send it to the service.

When the service responds to the message, the facilitator will identify the document type of the response from the service, identify the new state and the valid input documents for that state, and format an appropriate response for the client. The facilitator service can also pass the response through appropriate transformations, as shown in Figure 3. For example, if the client is an HTML browser, then the Facilitator service could return an HTML form prompting for appropriate input. Moreover, if the client is another service that can return a specification of its own service entry points, then the Facilitator service could automatically send the output message to appropriate client entry points; if a valid input document for the new state is returned, the Facilitator service could then forward it to the service, thus moving the conversation forward dynamically. As a result, the Facilitator service can help a client and service carry out an entire conversation without either the client or the service developer having to implement any explicit conversation control mechanisms. This means that the client developer does not need complete knowledge of all the possible conversations supported by all the services with which the client might interact in the future. Figure 3 sketches a flow chart illustrating an algorithm that a Facilitator service could execute each time it receives a message on behalf of a service.

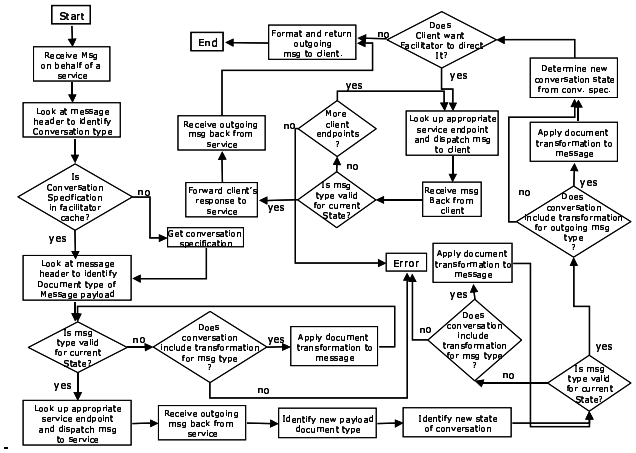

Fig. 3. A Facilitator service can handle messages on behalf of a service, including the application of document transformations to both incoming and outgoing messages.

3.1 Client Automation

An argument can be made that developers implementing Peer service clients will not want a facilitator service to direct their part of the conversation, both because they expect to hard-code the client parts of the conversation and also because they will find the idea of using a third-party to control conversation foreign[2]. However, decoupling conversation logic from business logic on the client side greatly increases the flexibility of a client by allowing it to interact dynamically with services even if their conversation policies do not match exactly. For example, the same client code could be used to interact with two services that support different conversation policies but common interfaces.

In order for a facilitator service to direct the client's part of a conversation, the controller must be able to dispatch messages the client receives from the server in order to generate documents that the server requests. This means that the client must be able to communicate its service interfaces to the Facilitator service. For example, the upper right hand corner of Figure 3 shows the Facilitator invoking the client.

3.2 Facilitator service state

The Facilitator service that we have outlined above does not include any performance management, history, or rollback mechanisms. If one subscribes to the idea that intermediate states of a Peer service's conversation are *not* transactional, and one also supposes that Conversation Management functionality (including performance history, status of ongoing conversations, etc.) is distinct from Conversation Control functionality, then the Facilitator service can operate in a stateless mode.

There are advantages and drawbacks to both a stateful and a stateless implementation. Services can easily transfer between stateless facilitators, even in the course of a single conversation, and a stateless facilitator does not have to handle recovery of ongoing conversation should it crash and restart. However, a stateful facilitator could be extended to explicitly support advanced features such as rollback mechanisms and visibility into conversational histories.

[2]Conversation with Kevin Smathers regarding conversation controllers, 1/4/2001.

4 Prototype implementation

We have implemented a prototype Facilitator service as well as example services and clients. We based our prototype on a previous implementation of a lightweight dynamic conversation controller that could receive messages on the behalf of the service, validate that each message was of an appropriate input document type for the current state of the conversation, dispatch each message to appropriate service entry point, and use the resulting output document types to identify the next appropriate interaction for the conversation [2]. We extended our previous architecture to include support for transformational interactions. Figure 4 sketches the components of our implementation.

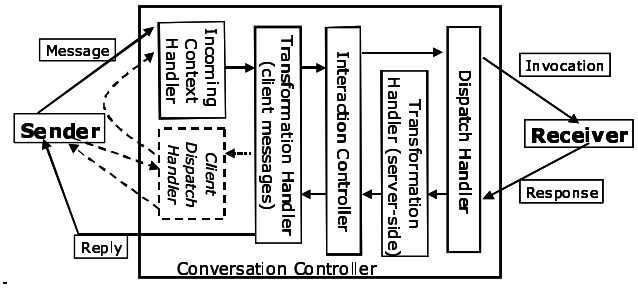

Fig. 4. The Facilitator handles messages on the behalf of services.

We tried to leverage as much existing technology as we could in building this prototype:

- We used the Web Service Conversation Language (WSCL) as a base for conversation specifications.
- We implemented the services as Apache JServ servlets (they could also have easily been implemented using E-Speak, Chaiserver, Jini, or CORBA).
- We used Apache's Xerces and Xalan packages for their XML and XSL functionality.
- We used a subset of the Web Services Description Language (WSDL) to describe Peer service interfaces. WSDL is a standard proposed by IBM and Microsoft.

We made a number of simplifying assumptions for this implementation. The Facilitator service is not responsible for maintaining management history, transactional guarantees, or the synchronization of multiple related conversations. We do not address security issues at this time.

4.1 Specifying Conversations and Transformations

As mentioned above, we extended the Web Service Conversation Language (WSCL) to meet our need for a conversation definition language that includes document transformation specifications. WSCL is an XML-based specification that defines a service interface in terms of a list of interactions (keyed by document type), a list of transitions that describe legal interaction orderings. We extended our usage of WSCL to include mappings of the input and output document types to corresponding document transformations. Space constraints do not permit a full description of

WSCL; instead we present a brief overview, first of WSCL and then of how we used it.

4.1.1 WSCL

WSCL addresses the problem of how to enable services from different enterprises to engage in flexible and autonomous, yet potentially quite complex, business interactions. It adopts an approach from the domain of software agents, modelling protocols for business interaction as *conversation policies*, but extends this approach to exploit the fact that service messages are XML-based business documents and can thus be mapped to XML document types. Each WSCL specification describes a single type of conversation from the perspective of a single participant. A service can participate in multiple types of conversations. Furthermore, a service can engage in multiple simultaneous instances of a given type of conversation.

For example, a service that supports the "secured album" conversation type from Figure 1 expects a conversation to begin with the receipt of a LoginRQ or a RegRQ document. Once the service has received one of these documents, then the conversation can progress to either a "logged in" state or a "registered" state, depending on the type of message the service generates to return to the client.

There are three elements to a WSCL specification:

- *Document type descriptions* specify the types (schemas) of XML documents that the service can accept and transmit in the course of a conversation.
- *Interactions* model the states of the conversation as document exchanges between conversation participants. WSCL currently supports four types of interactions: *Send* (the service sends out an outbound document), *Receive* (the service receives an inbound document), *SendReceive* (the service sends out an outbound document, then expects to receive an inbound document in reply), and *ReceiveSend* (the service receives an inbound document and then sends out an outbound document).
- *Transitions* specify the ordering relationships between interactions. A transition specifies a source interaction, a destination interaction, and a document type that triggers the transition. WSCL 1.0 also supports two special transitions: *Default Transition* and *Exception Transition.* A *default transition* is triggered if a valid inbound (for a *SendReceive* interaction) or outbound (for a *ReceiveSend* interaction) document is received for a given interaction, but no other transition is triggered. At most one default transition can be defined per source interaction.

Although WSCL specifies the valid inbound and outbound documents for an interaction, it does not specify how the conversation participants will handle and produce these documents. The WSCL specification of a conversation is thus service-independent, and can be used (and reused) by any number of services.

4.1.2 Associating interactions with transformations

With languages like WSDL and WSCL, the facilitator can act as a third-party conversation controller and use "reflected" XML specifications to direct services in their interactions according to a given conversational protocol without the service developers having to implement conversational logic themselves. However, WSDL and WSCL do not address the problem of how to facilitate conversation between two services that do not support a common conversational protocol.

For example, suppose a service that was only compatible with the "Secure Album" shopping cart conversation from Figure 1 wanted to converse with a service that was only compatible with the "Anonymous guest" shopping cart conversation. Also suppose there were an XSLT style sheet, *checkoutRQ2requestInvoice*, that transforms a *checkoutRQ* document into the document type *requestInvoice*, and another style sheet, *bill2invoice*, that transforms a *bill* document into type *invoice.* The facilitator could use these style sheets to facilitate conversation between the two services. However, we first needed to address the problem of how the facilitator can identify that style sheets exist and that it is appropriate for them to be used to mediate between the two conversation types.

There are a number of ways we could have addressed this issue. For example, we could have directly extended a conversation specification language to associate interactions with appropriate transformations. However, to do so would couple individual conversation specifications to each other. Alternatively, we could have made an assumption that there would be some third-party document transformation service available whom the facilitator could ask to transform the documents. However, for the purposes of this implementation we simply used a simple XML document to describe available transformations.

4.2 Directing conversations

In order to track the state of conversations, the Facilitator service needed to be able to perform the following functions:

- Given a message, determine the type of conversation (conversation specification) and the stage (interaction identifier) of the ongoing conversation.
- Given a message, identify its document type.
- Given a Conversation Specification and an interaction identifier from that specification, return the document types accepted as valid input to that interaction.
- Given a Conversation Specification, an interaction identifier from that specification, and a document type (representing an input document), return a Boolean indicating whether or not the document type would be accepted as valid input for that interaction.
- Given a Conversation Specification, a source interaction identifier from that specification, and a document type (representing an output document), return the target interaction represented by the transition from the source interaction given the output document type.

Our simple method of implementing the first two requirements was to give each message a special context element:

```
<Context>
   <ConversationId/>
   <In-Reply-To\>
```

```
    <Reply-With/>
    <DocumentType/>
</Context>
```

Each of these elements has an "owner" who controls the contents of the element value. The Facilitator service owns the ConversationId field, which can be used to map to the conversation type identifier, the current interaction and the valid input document types for the current interaction of the current conversation. The message sender owns the Reply-With and DocumentType element. The In-Reply-To element's value should be the value of the Reply-With element of the message to which the current message is responding. Each party is responsible for protecting the contents of their fields from tampering, e.g., using encryption.

To meet the last three requirements, each time the Facilitator service reads a new conversation specification (expressed in WSCL), it populates two hash tables: one that maps from interaction identifiers to valid input document types, and one that maps from source interaction identifier/transition document types to target interaction identifiers. The Facilitator service uses the first table to look up the valid input document types for a given interaction. It uses the second table to determine when a conversation has progressed from one interaction state to another (given the document type of an output document and a source interaction identifier).

4.3 Dispatching messages to services

In order to forward messages to appropriate service entry points, the Facilitator service needed to map input and output document types to service entry points. For this, we created a WSDL specification for each service. Each time the Facilitator service reads a WSDL specification, it populates two hash tables: one that maps from input document types to service entry point and output document types, and one that maps from output document types to service entry point and input document types.

One open issue is how to couple the dispatch and the conversation flow specifications. Our current implementation uses these tables to find actions that can handle incoming document types and produce output documents of appropriate document types. That is to say, currently the Facilitator service does not consider the state of the conversation when dispatching the message. This is because the WSDL and WSCL are completely independent. For the future, we are considering associating a WSDL specification with each interaction.

5 Related Work

A number of peer-to-peer systems support the coordination of distributed applications. However, to the best of our knowledge existing systems require the distributed process to be centrally managed or else require participants to download and install platform specific components, or both. For example, the SETI project [11], which allows anyone with a computer and an internet connection to donate spare computing cycles to search for extraterrestrial life, consists of an application with specific hard-coded functionality that participants download; a central coordinating site combines the results from the test sites. Parabon Computation's Frontier product [12] allows clients to download code dynamically, but still requires developers to write separate code to handle the central control and monitoring of the distributed clients. Consilient Sitelets [13], which bundle up project-related data from the documents to the workflow, do not require central coordinating software, but do require participants to download and install platform and application specific components. Our facilitator requires neither central management nor installation of components; we require only that participants support XML message communications (e.g., SOAP).

Several existing agent systems allow agents to communicate following conversational protocols (or patterns). However, to the best of our knowledge, all of these are tightly coupled to specific agent systems, and require that all participating entities must be built upon a common agent platform. For example, the Knowledgeable Agent-oriented System (KaoS) [14] is an open distributed architecture for software agents, but requires agent developers to hard-wire conversation policies into agents in advance. Walker and Wooldridge [15] address the issue of how a group of autonomous agents can reach a global agreement on conversation policy; however, they require the agents themselves to implement strategies and control. Chen, et al. [16] provide a framework in which agents can dynamically load conversation policies from one-another, but their solution is homogeneous and requires that agents be built upon a common infrastructure. Our facilitator requires only that a participating service produce two XML-based documents – 1) a specification of the conversational flows it supports and 2) a specification of the service's functionality (describing how the service can be invoked). In addition, our transformational interactions are unique in that they enable the facilitator to negotiate appropriate document transformations so as to mediate between mismatched conversation definitions automatically.

Some E-Commerce systems support conversations between services. However, these all require that the client and service developers implement matching conversation control policies, and to the best of our knowledge, none provide explicit support for document transformations. RosettaNet's *Partner Interface Processes* (PIPs) [10] specify the roles and required interactions between two businesses. *Commerce XML* (cXML) [17] is a proposed standard being developed by more than 50 companies for business-to-business electronic commerce. cXML associates XML DTDs for business documents with their request/response processes. Both RosettaNet and CommerceXML require that participants pre-conform to their standards. Our work is completely compatible with such systems, but is also unique in that we do not require the either the client nor the server pre-conform to any given conversational specifications.

Insofar as they reflect the flow of business processes, service conversations also resemble workflows. However, as the authors of the E-Speak Conversation Definition Language (CDL) [18] observe, workflows and conversations

serve different purposes. Conversations reflect the interactions between services, whereas workflows delineate the work done by a service. A conversation models the externally visible commercial interactions of a service, whereas a workflow implements the service's business functionality. In addition, workflows represent long-running concurrent fully integrated processes, whereas E-Service conversations are loosely coupled interactions.

6 Conclusions and Future Work

We have proposed here a mechanism for a facilitator service that can act as a proxy to services, enabling peer services to engage in complex interactions with a minimum of developmental effort. Our solution is unique in that we distinguish between the conversational protocols and service-specific interfaces. This allows us to provide an extremely lightweight solution relieving service developers from the burden of implementing conversation-handling logic. In addition, we also introduce *transformational interactions* that allow facilitator services to leverage document transformations and make possible the automated coordination of complex conversations between peer services that do not support compatible message document types.

In the future, we plan to investigate more sophisticated uses of conversation policies. For example, we would like to provide a model for the explicit support of deciding conversation version compatibility. We would also like to explore how to support both nested conversations and multi-party. Finally, we hope to address how to exploit document type relationships when manipulating message documents. For example, we would like to use subtype polymorphism to establish a relationship between a document type accepted as input by an interface specification and a corresponding document type in a conversation specification.

References

[1] Jeff Schneider, "Convergence of peer and web services," http://www.openp2p.com/pub/a/p2p/2001/07/20/convergence.html, July 2001.

[2] Harumi Kuno and Michael Lemon, "A Lightweight Dynamic Conversation Controller for E-Services," in *International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, June 2001.

[3] Web page, "Is p2p right for b2b?," http://www.cisco.com/warp/public/750/iq/gen/fea/col/gen_fea_col_0018/gen_fea_col_0018_1.shtml.

[4] Web page, "What can p2p do for b2b?," http://ecommerce.inetemet.com/outlook/article/0,1467,7761_486031,00.html.

[5] Harumi Kuno, "Surveying the E-Services Technical Landscape," in *International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, June 2000.

[6] Ariba, Inc. and International Business Machines Corporation, "UDDI Technical White Paper," Tech. Rep., Ariba, Inc. and International Business Machines Corporation and Microsoft Corporation, Sept. 2000.

[7] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana, "Web Services Description Language (WSDL) 1.0," Sept. 2000.

[8] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, and Scott Williams, "Web Services Conversation Language (WSCL) 1.0," Tech. Rep., Hewlett-Packard Web Services Organization, May 2001.

[9] Dorothea Beringer, Harumi Kuno, and Mike Lemon, "Using WSCL in a UDDI Registry 1.02: UDDI Working Draft Best Practices Document," Tech. Rep., Hewlett-Packard Company, 2001.

[10] Web page, "http://rosettanet.org," .

[11] Web page, "Seti institute online," http://seti.org.

[12] Web page, "Parabon computation, inc.," http://parabon.com.

[13] Web page, "Consilient," http://www.consilient.com/.

[14] Jeffrey M. Bradshaw, "KAoS: An Open Agent Architecture Supporting Reuse, Interoperability, and Extensibility," in *Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1996.

[15] A. Walker and M. Wooldridge, "Understanding the emergence of conventions in multi-agent systems," in *First International Conference on Multi-Agent Systems*, 1995.

[16] Qiming Chen, Umesh Dayal, Meichun Hsu, and Martin Griss, "Dynamic Agents, Workflow and XML for E-Commerce Automation," in *First International Conference on E-Commerce and Web-Technology*, 2000.

[17] Web page, "cxml.org," http://www.cxml.org.

[18] HP E-Speak Operations, "Conversation Definition Language Specification for UDDI, Version 1.0," Tech. Rep., Hewlett-Packard Company, Nov. 2000.