Published on **dev2dev** (http://dev2dev.bea.com/)
http://dev2dev.bea.com/pub/a/2004/12/ebXML.html
See this if you're having trouble printing code examples

Introduction to ebXML

by Blake Dournaee
12/06/2004

Abstract

This article will provide the reader with a general introduction to ebXML, including an overview of the main goals of the specification set and a conceptual outline of why ebXML exists. In addition to the conceptual overview, the reader will be given an introduction to some of the details regarding the messaging layer, registry, business policies, and the relationship of ebXML to XML Web services.

Introduction to ebXML

Why do we need another XML language? It seems like every day a new XML standard is being developed or ratified. It is like a giant XML steamroller is pressing language after language on both industry and the public alike. This relentless condition causes a certain type of scrambling, wherein technologists must try to ascertain the value of moving from whatever legacy format they used in the past to the new, glorified XML version. This must all be done, of course, while maintaining sanity and security.

This never-ending flow of XML languages is partly due to the simplicity of XML itself. XML cannot accomplish much on its own, and in some ways it is too simple. Specification upon specification must be developed to mold XML into something useful. At times, it appears that new XML languages are developed just for fun, as if there is a constant need to "XMLify" every possible computing format or interaction.

The main goal in this introduction is to convince the reader that ebXML is not just a blind format change from non-XML EDI (Electronic Data Interchange) based business interactions. While it may seem as though ebXML has joined the ranks of "just another XML language," in fact it has important conceptual benefits for business, mainly due to its ad hoc vision.

The Vision of Ad Hoc Business

One of the most powerful features of ebXML is its ability to achieve *ad hoc business interactions*. At first glance, the term *ad hoc* may conjure up negative images of unplanned disasters and unexpected events, but it is precisely this feature of ebXML that makes it especially powerful for conducting electronic business.

To use a simple analogy, consider buying groceries at the local supermarket. Suppose you always buy groceries on a regular basis from supermarket A. Over time, you develop a business relationship with the supermarket based on the goods provided and the process undertaken to purchase items, such as interacting with the clerks and using your ATM card to make purchases. While you have a long relationship with supermarket A, your relationship is ad hoc and impromptu.

Consider now a new store, supermarket B, that opens with lower prices for the same goods. It is in your best interest, of course, to begin a new business relationship with supermarket B, and you can do so because the business processes and interactions are expected to be the same, that is, you'll speak in English and probably use your ATM card to make the transactions happen. Here, the ad hoc

nature of the business relationships is what makes the free market economy work; you can easily abandon your previous relationship with supermarket A and quickly make a new relationship with supermarket B.

Electronic business over the Internet, however, has infrastructure costs that factor in to the overall price of doing business. For example, if two businesses arrange to trade electronically, they must pay the price of arranging the infrastructure and software, as well as formalize the business interactions and policies, including adequate security policies.

Over time, if new businesses emerge that offer lower prices for goods and services, the mode of interaction and the technologies required become an economic barrier. In other words, if the cost of doing business involves hefty infrastructure and process modification costs, then there is no reason to switch to a lower price of goods if it isn't going to matter in the end.

For example, suppose supermarket B offers slightly lower prices, but the clerks all speak a different language, and they only accept currency in two-dollar bills. In this case, the extra costs required to do business, such as learning the language and having the correct currency in hand, may outweigh the price difference.

One of the core values of ebXML is its vision of ubiquity from a technology perspective. It is built around XML, SOAP, HTTP, and SMTP, all open standards with low barriers to entry. In theory, the focus on technological ubiquity should allow electronic businesses to approach the ad hoc free market concept that we all experience when we shop at the supermarket.

What About XML Web Services?

It may seem as though I have committed a crime of sorts, mentioning SOAP and XML without alluding to XML Web services or service-oriented architectures. It turns out, however, that the ebXML architecture predates many of the general XML Web services standards but stays true to most of the concepts. One simple way of understanding the breadth of both XML Web services and ebXML architectural concepts is to rally around three terms: *wire*, *description*, and *discovery*.

The first term represents technologies for message transport. For both XML Web services and ebXML this is SOAP, but the similarities don't go much further than that. XML Web services has a loosely coupled wire stack that consists of separate specifications for reliable transport (WS-Reliability) and security (WS-Security) while ebXML rolls all this functionality into its messaging standard, ebMS, using a mix of technologies.

For the description and discovery stacks, XML Web services uses the Web Services Description Language (WSDL) and UDDI, respectively. For ebXML, these description and discovery mechanisms are part of the ebXML registry; further, ebXML includes additional specifications for business process and collaboration.

In short, ebXML is a self-contained set of specifications that is internally consistent and doesn't rely on emerging standards and specifications.

ebXML Overview

To achieve the ad hoc vision just discussed, ebXML provides a complete framework for business interactions, all delivered as a set of vendor-neutral specifications. This complete framework is designed to answer a number of holistic business questions. The perspective for these questions is framed with respect to a given trading partner:

- How do I describe my business process and specific interfaces?

- How do I share my business process with other partners?
- How do I find out which business processes my partner supports?
- How do I describe the business messages for a particular transaction?
- How do I describe the security policy and technical configuration to be used?

In theory, if a trading partner can describe itself in these terms, it can move one step closer to involving itself in an impromptu, electronic free market. Many of these questions can be answered by implementing a shared registry of information where business agreements and processes can be centralized. This central point repository is known as the ebXML registry. Along with the registry, there are specifications for the actual wire level messaging layer as well as for business process specifications and collaboration information. The set of concrete ebXML specifications maps to these concepts as follows:

- Centralized Shared Registry: *Registry Information Model, Registry Services Specification* (ebRIM, ebRS)
- Business Processes & Collaboration: *Business Process Specification Schema, Collaboration-Protocol Profile and Agreement Specification* (ebBPSS, ebCPPA)
- Messaging: *Message Services Specification* (ebMS)

ebXML Registry

A vital part of an ebXML implementation is the ebXML registry. The registry itself is quite versatile and capable of representing a large range of data objects including XML schemas, business process descriptions, ebXML Core Components, UML models, generic trading partner information, and software components. In order to support such a wide variety of data, the ebXML registry is designed more like a database, with a well-defined information model rather than a directory. This point is important because there is a prevailing belief that the ebXML registry is in competition with registry services for XML Web services such as UDDI. In fact, the two have different purposes entirely: one might find a published ebXML registry *endpoint* in a UDDI directory, but UDDI isn't designed to handle the complex classification relationships possible with the ebXML registry.

There are two ways to look at the ebXML registry: from the outside looking in, or from the information model looking outward. The former view provides more of a simple overview because it is from the viewpoint of a client looking to access one of the two interfaces provided by the ebXML registry. These two interfaces include the *Lifecycle Management Interface* and the *Query Management Interface*. The LifeCycle Management Interface is used to manage the lifecycle of the objects (also called *repository items*) in the registry, and the Query Management Interface is used to make queries against a registry. In order to grasp how the interfaces work, we must take a brief look at how information is logically stored within the registry itself.

ebXML Registry Information Model

The core information model used by the ebXML registry is a tree-based classification scheme, which means that information about industries or business partners is arranged in a hierarchy. One major difference between the ebXML registry information model and a simple hierarchy is its ability to convey more complex relationships. For example, consider the following tree-based hierarchy presented in the ebRIM specification:

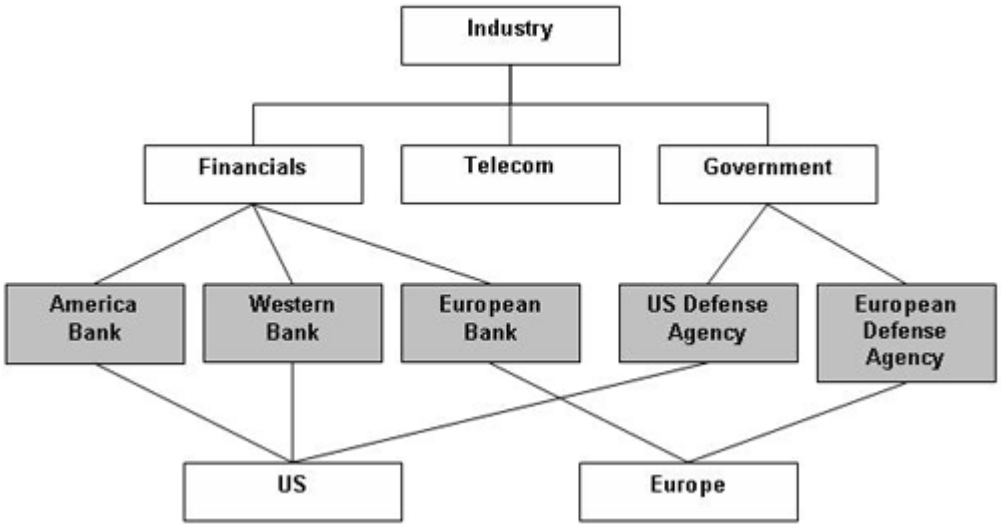

Figure 1. ebXML Registry Information Model (ebRIM)

In Figure 1 the reader should note that a portion of the hierarchy is shaded. The shaded portion refers to the actual registry objects, while the unshaded portions are called *classifications*. In many ways, the classification scheme used by the ebXML registry behaves more like an *ontology* or structure of knowledge. Notice that arbitrary leaves in the tree can participate in additional classification relationships.

ebXML Registry Interfaces

The ebXML Registry Architecture is defined in terms of the registry service and the registry client. The registry service has two main interfaces for managing objects in the information model: lifecycle management and query management. The lifecycle management interface has abstract methods such as `submitObjects`, `updateObjects`, `removeObjects`, and `deprecatedObjects`, which are used to submit objects or classifications to the information model. Similarly, the query management interface has interfaces such as `submitAdhocQuery`, `getRegistryObject`, and `getRepositoryItem`, which are used to query the registry itself.

The abstract registry service interfaces are defined using a Web Services Description Language (WSDL) file, available from the OASIS ebXML Registry Technical Committee. There are three concrete bindings to the two interfaces: SOAP over HTTP, ebMS, and straight HTTP. This diversity with binding choices means that ebXML clients will come in the form of both thin clients and fat clients. The thin clients are likely to be browser-based, read-only interfaces, and the fat clients are used for making changes or additions to a running registry.

To summarize then, there are five important ebXML specifications: ebRIM, ebRS, ebBPSS, ebCPP, and ebMS. The ebMS specification defines the ebXML Message Service Protocol and is designed to enable the secure and reliable exchange of business messages between trading partners. While the actual business messages sent as part of a business transaction are indeed important, the messaging portion of ebXML is just a small part of the overall ebXML architecture, which is defined in terms of many different components and specifications. An important high-level picture of how an ebXML interaction occurs can be framed in terms of ebXML *functional phases*.

ebXML Functional Phases

The act of arranging for a new business relationship means accessing the shared ebXML registry, which is generally governed by the current functional phase. Three functional phases are defined by the ebXML technical architecture. These include the implementation phase, discovery and retrieval phase, and runtime phase. Each phase carries with it its own security requirements and processes.

The next three subsections give a quick overview of each phase. In general terms, the first two phases of implementation and retrieval represent a handshake of sorts, while the final runtime phase represents the actual units of business.

Implementation Phase

The implementation phase of ebXML is considered the time when a trading partner is making an active decision to do business using the ebXML framework. As shown in Figure 2, in this phase the trading partner will analyze its business processes in terms of the generalizations provided by ebXML and will publish its business processes to a registry. During this phase, an actual ebXML implementation must be produced, either built in-house from the core ebXML specifications or obtained from a third-party vendor. The result of the implementation phase is a working ebXML framework including a set of published business processes and interfaces. The Collaboration Protocol Profile (CPP) is also published at this time.

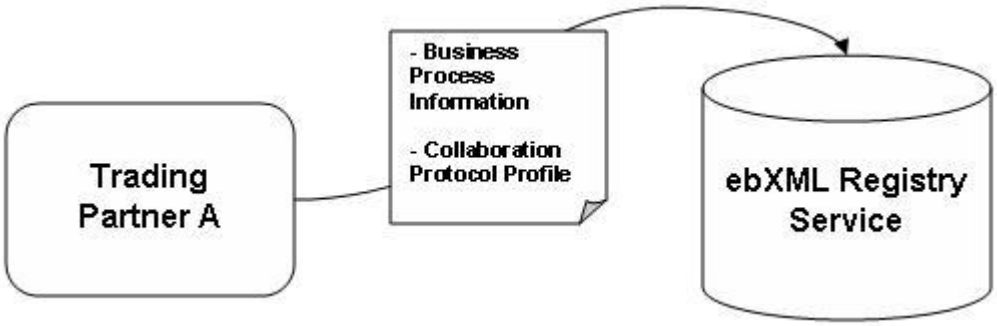

Figure 2. Implementation phase

Discovery and Retrieval Phase

The discovery and retrieval phase of ebXML involves trading partners using the registry to discover business processes and interfaces published by other trading partners. Typically, the CPP for a specific partner or set of partners is exchanged at this time. The CPP describes specific business processes and technology details, including security, transport, and reliability details. The specific details denoted in the CPP are used as a basis for messages exchanged during the runtime phase. Figure 3 shows two trading partners discovering each other's Collaboration Protocol Profile documents.

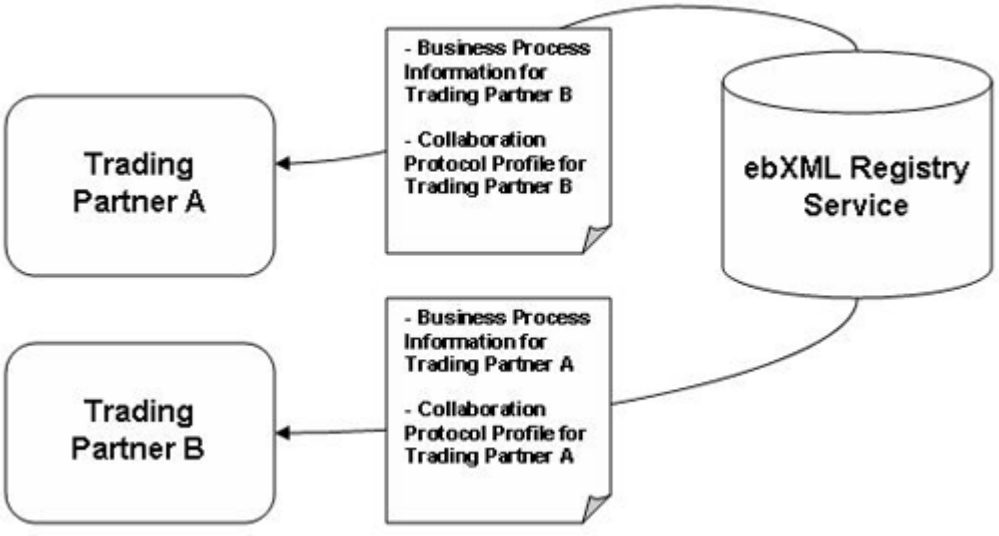

Figure 3. Discovery and retrieval phase

Runtime Phase

The runtime phase is concerned with the actual business transactions and choreography of messages exchanged between partners. Typically, there is no runtime access to the registry during the runtime phase. The CPP instances published by each participating trading partner are narrowed to form the Collaboration Protocol Agreement (CPA). The CPA is a special business agreement tied to a specific transactional conversation and makes explicit requirements derived from the intersection of the various CPP instances published by each of the trading partners. As shown in Figure 3, each trading partner derives the CPA by performing an intersection between each of the partner's CPP instances. Before actual ebMS messages are exchanged, each trading partner should compare CPA instances to ensure consistency between the two. The CPA instances should match on both ends before the transaction occurs.

Figure 4 can be summarized in three simple steps. In step 1 each trading partner is responsible for obtaining the necessary CPP document for the business partner it would like to engage. In most cases the CPP will be retrieved from an ebXML registry. In step 2, each partner derives the Collaboration Profile Agreement (CPA), which makes explicit the range of choices offered in the CPP. Finally, in step 3, the partners can begin business transactions under the governance of the CPA. In this sense there is a policy relation between the ebMS messages and the derived CPA.

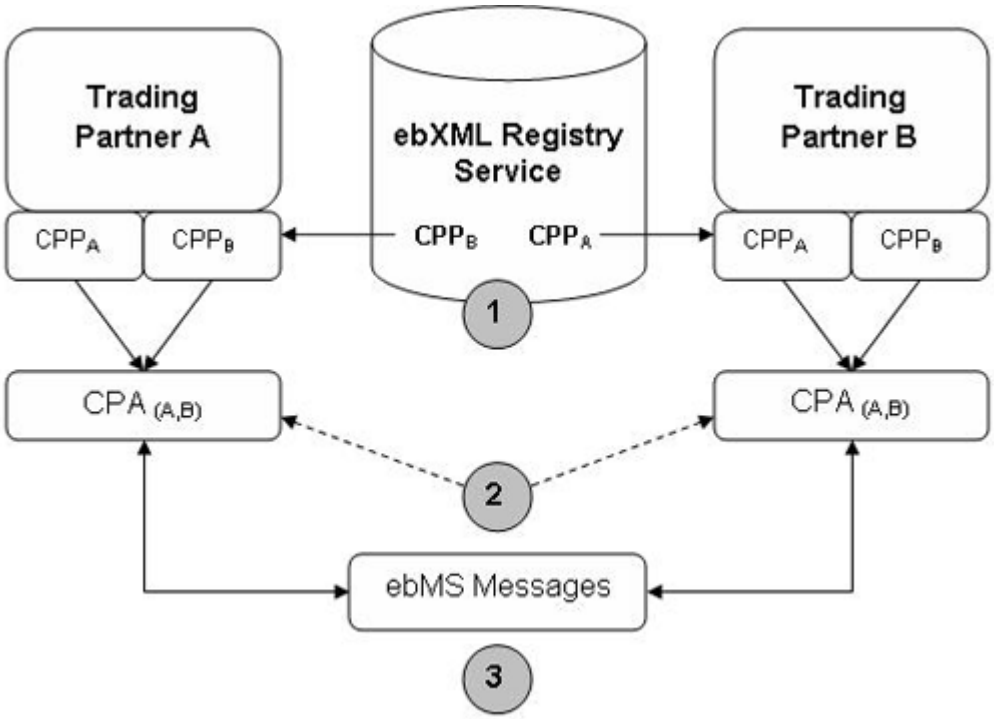

Figure 4. Runtime phase

Collaboration Protocol Profile and Collaboration Protocol Agreement

As described earlier, one of the key components of ebXML is an artifact known as the CPP or Collaboration Protocol Profile. The CPP contains specific technology implementation details in support of a particular business process. The CPP is published to the ebXML registry and outlines supported technology binding details. The main purpose of the CPP is to ensure interoperability between trading partners relying on the ebXML framework from possibly disparate vendors.

The CPP is important from a security perspective because it houses security policy information along with specific message exchange details, all represented in XML. In the context of a security gateway or firewall, this policy information can be used to configure the security proxy to handle the appropriate message-level security operations.

Some of the message exchange details defined by the CPP include specifics such as transport protocol mechanisms, message reliability mechanisms, transport security mechanisms, trust artifacts such as X.509 certificates, and message-level security policy information. In addition to implementation details, the CPP also refers to the set of supported business collaborations that define the supported business transactions.

A CPP alone doesn't enforce specific choices for a message exchange. Only the intersection of at least two CPP documents produces a CPA capable of enforcing specific message-level security and reliability mechanisms. The organization of the CPA is nearly identical to the CPP, and for the purposes of declarative security policy, the elements and descriptions that appear in the CPP are shared by the CPA. In the end, the reader should think of the CPA as the final governing security policy statement for a set of ebMS message exchanges.

CPP Structure

The CPP describes the set of message exchange capabilities and business collaborations supported by a trading partner. Message exchange capabilities are the implementation details and policies for the runtime phase, and business collaborations are specific business transactions, which are described by process specification documents. Figure 5 is a pictorial view of the CPP.

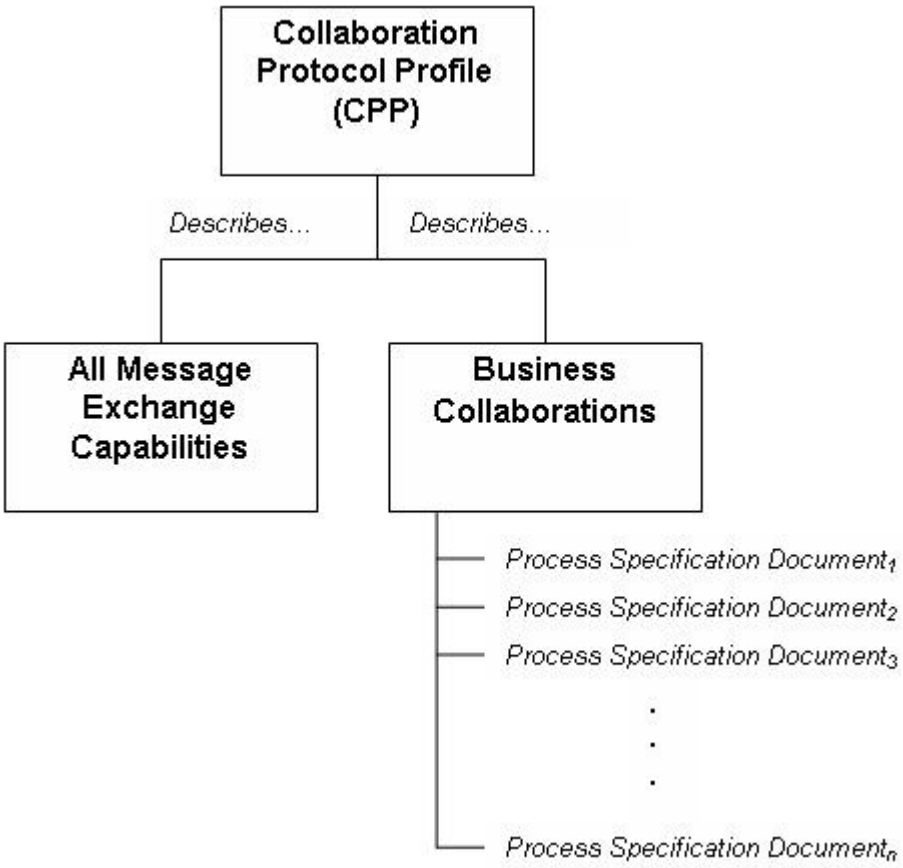

Figure 5. CPP concepts

The most important point regarding Figure 4 [JS: should this be Figure 5, not 4?] is the fact that the CPP represents a list of choices that will be narrowed down when the CPA is generated. The CPA is generally tied to a single process specification document used by both trading partners. The process specification document represents the unit of business being conducted and is largely out of scope for this article. More information regarding the process specification document can be found in the ebXML Business Process Specification Schema.

CPP Data Model

The concrete CPP instance is defined by five direct child elements, shown in Listing 1. A simple BNF grammar is used to describe the cardinality of the elements. A "+" means one or more, a "?" means zero or one, and a "*" means zero or more. The absence of an indicator means exactly one.

Listing 1. The Collaboration Protocol Profile (CPP) XML structure

```
<CollaborationProtocolProfile>
  (<PartyInfo>)  +
  (<SimplePart>) +
  (<Packaging>)  +
  (<Signature>)  ?
  (<Comment>)    *
</CollaborationProtocolProfile>
```

Each element is described as follows:

- `<PartyInfo>`
  This element identifies the organization (or its parts) whose capabilities are described by the CPP.

- `<SimplePart>`
  This element describes the components used to make up composite messages.

- `<Packaging>`
  This element is used to describe how the message header and payload are packaged for transmission.

- `<Signature>`
  This element contains an XML Signature used to sign the actual CPP document.

- `<Comment>`
  This element is used for comments.

Each of the child elements of the `<CollaborationProtocolProfile>` contains many more nested children. The CPP itself is quite complex, and an entire article could be devoted to examining the structure. For now we will stop at the outermost layer to give the reader a high-level overview.

If we were to move away from the concepts and go a level deeper to the concrete implementation, we could look at each of the functional phases in more detail.

Implementation, Discovery, and Retrieval Phase Details

From the developer viewpoint, these phases would entail interaction with an ebXML registry for publishing and retrieval of business documents and processes. For the implementation phase, the ebXML registry *lifecycle management* service would be used, and for the discovery phase the *query management* interface would be used. To actually access these interfaces, the developer has a choice of three concrete bindings against the published WSDL. The WSDL in this case is part of the ebXML Registry Service specification and will differ depending on the version of the specification. The available bindings include SOAP over HTTP, ebMS, and straight HTTP.

The first binding would represent a fat client where the developer builds an actual client using a WSDL description for managing the state of objects in the registry. In this case, the client behaves

like a pure Web services client. The second binding is very similar to the first, but the wire format uses ebMS (which is built around SOAP). The third option is an HTTP interface, which is the most lightweight of the three. In principle, it could be implemented with an XML-aware browser. Using this binding the developer may be accessing the registry through a Web application native to the registry itself. It must be stressed here that there are no business transactions at this stage; we are simply managing the registry by providing the necessary information for other partners engaged in business transactions.

Runtime Phase

Once the business handshaking has occurred in the first two phases, the actual messages transmitted are governed by the ebMS specification. The ebMS specification uses SOAP with Attachments to package business data. The main SOAP payload contains signed header information using XML Signature, including a manifest or list of the actual payloads. The actual business message is conveyed as one or more MIME parts and is not represented in the SOAP body. While ebMS is based around SOAP, it uses SOAP as a convenient packaging mechanism rather than a full fledged Web services transport.

The schema definition for ebMS is defined in terms of SOAP v1.1 extension points. In particular, a BNF grammar style view of the element structure is shown in Listing 2. Compared to SOAP, the ebMS specification adds a `<MessageHeader>` in the SOAP header and a `<Manifest>` element in the SOAP body. This structure uses standard cardinality symbols where "*" means zero or more, "?" means zero or one, "+" means one or more, and the absence of a symbol means exactly one. Namespaces have been omitted for clarity.

Listing 2. SOAP/ebMS element structure

```
<Envelope>
  <Header>
    <MessageHeader>
      <From>
      <To>
      <CPAId>
      <ConversationId>
      <Service>
      <Action>
      <MessageData>
      (<DuplicateElimination>) ?
      (<Description>)          *
    </MessageHeader>
  <Header>
  <Body>
     (<Manifest>
       (<Reference>        +
         (<Schema>)        *
         (<Description>)   *
        </Reference> )     +
     </Manifest> )         ?
  <Body>
</Envelope>
```

The actual business payload is domain-specific and would be found as a MIME attachment after the first main SOAP payload.

Conclusion

This article introduced ebXML and its vision of ad hoc business. At this point the reader should have

a good *conceptual* view of ebXML, including an overview of the constituent standards. In addition, the reader should have a grasp of the purpose of the Collaboration Protocol Profile (CPP) and how it differs from the Collaboration Protocol Agreement (CPA), as well as some insight into the ebXML functional phases.

Resources

- ebXML.org
- S/MIME Specification
- PGP/MIME
- W3C XML Signature
- W3C XML Encryption
- W3C Web Services Activity
- SOAP/XML Protocol
- SOAP Messages with Attachments
- OASIS WS-Security

Blake Dournaee is a senior architect at Sarvega, Inc, based in Oakbrook Terrace, IL.

Return to dev2dev.