IxRetail Infrastructure Study

Ron Kleinman

October 23, 2000

Introduction

This study will enumerate nine major infrastructure options available to the IxRetail committee, based upon a study of similar existing vertical industry XML standards.

Once we have established the range of the “possible”, we can then begin to pare things down by applying use cases generated in the other working groups to determine the actual requirements our infrastructure must satisfy.

Overarching Issues

There are two key issues that must be addressed up front, which are not purely technical. The first impacts the effort to develop the IxRetail infrastructure while the second determines what the value proposition of that infrastructure will be after it is completed.

1. Infrastructure Levels: Make vs. Take

We have an advantage over earlier XML standard efforts in other industries, since several “horizontal” standards are under development, and we can leverage off the functionality they provide.

A typical XML infrastructure for a standard such as IxRetail is composed of the following layers, and “make vs. take” decisions must be made for each.

A. Transport

This layer functions as the delivery mechanism for XML messages.

Possible choices are HTTP, HTTPS, SMTP, FTP, or a “tightly coupled” OO based protocol such as Corba’s IIOP or DCOM’s DCE.

There is clearly no need to “make” this layer. The challenge is to select the established transport that best fits our needs. In fact, our eventual choice of a transport will likely be dictated by the decisions we make at the envelope layer.

B. Envelope

The envelope layer provides routing, packaging, encoding and security capabilities for XML messages.

Possible choices are SOAP, ebXML (TRP), XP (the W3C's attempt to define an envelope layer, based around an initial SOAP v1.1 submittal) or a proprietary message oriented middleware product such as MSMQ or MQ/Series.

Given recent developments within the various horizontal XML standards bodies, there is no longer any need for us to "make" this layer. The real challenge is choosing the optimum one to "take", especially considering the rapid evolution of the possible choices identified above.

Further analysis of the current functionality offered by each of these options is probably premature at this time.

C. Header

The Header must define a set of message "types" which support the specific requirements of the IxRetail standard, as determined by use case analysis. Many of the possible requirements will be examined in more detail below. They include, but are not limited to:

Session setup and maintenance of session state
Guaranteed message delivery
Matching Requests with asynchronous Responses
Support for an asynchronous Publish / Subscribe capability
Support for transactions and transaction choreographies

There is no widely supported vendor-neutral solution available for many of these requirements, although several are currently supplied by existing proprietary messaging products. It appears at the present time, that if IxRetail is to adequately support this level of functionality in a non-proprietary fashion, we unfortunately must "make" at least some of this Header level ourselves.

2. Single "Reference" Infrastructure (Y/N)

This issue relates directly to the requirement for "interoperability".

YES:

If we chose to define a single "reference" IxRetail infrastructure (spanning all three layers defined above) which an application must be capable of using if it is to be awarded the “IxRetail-compliant” label, then we greatly enhance the chances that any two such IxRetail-compliant applications will interoperate “out of the box”.

The existence of such a reference infrastructure does NOT necessarily mean that two applications must use it whenever they communicate... they can use any mutually agreed upon substitute without affecting interoperability. What it does mean is that if an alternative infrastructure cannot be negotiated or configured, both parties must be capable of supporting the IxRetail reference infrastructure.

Note that any alternative infrastructure normally must supply at least the same functionality as the reference. For example, if the reference supplied data authentication and encryption by using HTTPS as the underlying transport, then any alternative infrastructure used at a site requiring such data security must supply them as well.

Many XML vertical standards (SIF, OTA, HitisX, UCCnet) already include such a reference infrastructure.

No:
The alternative approach would leave at least part of the infrastructure unspecified, thereby reducing the level of interoperability provided by IxRetail, and greatly complicating (if not making impossible) full compliance testing. An analogy could be drawn with two EDI applications, which while agreeing upon the exact format of a purchase order, were unable to interoperate because they were deployed on two different VANs.

As I really can’t think of a good reason to chose this option other than simple expediency, I leave it to others to make a better case for not specifying a reference IxRetail infrastructure.

The remaining issues concern purely technical infrastructure trade-offs, mostly within the Header layer. The first of these has the most far reaching effects.

3. Closely Coupled vs. Loosely Coupled XML

A given XML-based standard has many viable alternatives for the degree of “coupling” it will impose between any two applications which use it to interoperate. Each offers a different trade-off between two very important design considerations.

A. Value of Compliance

This determines how meaningful the “IxRetail-compliant” brand on a retail application will be for the end user. Here, the tighter the coupling option chosen, the better.

B. Ease of Migration

This determines how easy it is to migrate from an existing legacy enterprise to one which is primarily composed of IxRetail based applications. Here, looser coupling is better.

There are four major alternatives which will be examined below, in order of decreasing “tightness”. They range from using XML to either:

1. Specify remote procedure calls,

2. Define the document exchanges (of varying rigidity) between a set of pre-specified application partners.

3. Define a set of data objects which may be shared among an open set of cooperating applications.

A. Communication via Function Calls (XML RPC)

Here applications exchange procedure calls using XML to cross firewalls, basically extending Corba or DCOM to the Internet, only not as well (since RPC provides a procedural rather than an object-based interface). This is a very tight XML coupling choice that seems wrong for IxRetail on several counts.

1. IxRetail had its origins in the ARTS Data Model, not a UML specification. It is UML that maps well to XML RPC. Data models are normally mapped directly to a standard based upon XML document exchanges, rather than procedure calls.

2. We are assuming a retail Enterprise Intranet environment which shares a common security policy. Corba or DCOM would therefore provide a better tightly-coupled solution than using XML to convey RPCs.

B. Rigid Message Schemas, Rigid Partners

This option requires us to eventually produce the complete definition for the set of all documents exchanged between all IxRetail compliant applications within the retail enterprise.

Each application is assigned a schema for each message it will send or receive, with all XML tags mandatory, and each partner clearly identified.

Advantages

The IxRetail specification is "tight". An application is deemed compliant only if it sends and receives valid XML messages in a set of approved "message exchange choreographies".

Coupled with a selection of a 3-level reference infrastructure, this option almost guarantees that a set of IxRetail compliant applications would interoperate, offering significant value to the end user.

An IxRetail-compliant enterprise could therefore support "plug & play" retail applications in the same way that a UnifiedPOS-compliant application supports plug & play POS peripherals. The retail application model is complete.

> "Write an IxRetail-compliant application once, and deploy it in any enterprise which supports IxRetail."

Disadvantages

You have to pay for paradise. The analogy with POS devices is revealing. The current UnifiedPOS specification supports about twenty-five COMPLETELY DEFINED device types. Any supported peripheral MUST be assigned to one of these types, or it cannot be supported.

Mapping this model to the set of applications in a retail enterprise poses one great problem.

> Is it even possible to define the various applications comprising a retail enterprise to this level of detail?

The application boundaries in existing retail enterprises vary widely. Any attempt to "exactly" define the format of the data exchanged between the various types of applications (Pricing / POS / Customer Loyalty / Merchandising / Order Fulfillment / Property Management / Customer Resource Management) is likely to:

1. Not match what the majority of deployed legacy applications do today

2. Not meet the specific needs of a significant fraction of the retail enterprises which form the primary user base for this standard.

Consider that the attempt to put business rules on the wire ran into exactly the same overspecification problem, even though it represented a looser application coupling than we are dealing with here.

As a result, migration to the IxRetail standard becomes a formidable task, because ALL direct partners in an enterprise have to be replaced by their IxRetail-compliant equivalents before ANY IxRetail-compliant application can work as intended.

Conclusions

Only where there is strong consensus around exactly what data a particular “type” of retail application supplies and needs to receive, can this tight an approach be used.

Perhaps we can reach this level of specification for a few retail subsystems... but that is a matter for the individual Working Groups to decide, based upon how closely the boundaries of their assigned subsystems map across existing (and diverse) retail enterprises.

C. Loose Schema, Rigid Partners

Enter the “optional” XML entities. Here we still define a set of retail application “types” and a corresponding set of messages which each application type exchanges, but not all data fields in these messages are required.

Advantages

IxRetail application boundaries become less rigid, so the standard can more easily be supported by existing applications. In addition, an “extension” XML element with a deliberately undefined internal format can be inserted at key points in the message schemas, to allow for future expansion.

Subsequent IxRetail versions could then add new functionality by completely defining such an extension in an updated schema, while still keeping the altered messages “valid” from the point of view of applications running older versions of IxRetail.

Disadvantages

There are several disadvantages to this approach.

Any optional field poses the problem that it cannot be counted on to be present. How can an application adjust if its partners do not supply a needed (but optional) field?

One answer involves using a variant of Trading Partner Agreements (TPAs), namely “Application Partner Agreements” (APAs) to specify which optional fields are and are not supported. These can be exchanged within the IxRetail Message set or can be externally matched by a local system integrator.

But whichever matching technique is used, optional fields eliminate the possibility of plug & play components (similar to those of JavaPOS and OPOS) without some level of application-specific negotiation. This reduces the value of the IxRetail-compliant brand, as “out of the box” interoperability is no longer likely.

Finally, even with optional fields to “loosen” the schema, the requirements placed upon an IxRetail-compliant application to send and receive a specific set of XML messages with a set of specified partners, may still be too tight for widespread adoption.

Conclusions

Allowing optional XML entities, except within a predefined “extension” element, does not seem an attractive alternative for the IxRetail standard.

D. Looser Schema, Undefined Partners

This option completely replaces the document exchange models described above, with one based upon data exchange.

The previous requirement to carve a retail enterprise up into a set of predefined application “types” is removed. The boundary between applications is blurred to the point where an application need no longer depend upon the presence of a specific type of partner to permit the exchange a set of predefined messages. Rather each application interface is based totally upon the data elements it provides and the data elements it needs.

We have been calling this the “Alice Model”, and it is quite different from the previous options discussed above. A rather lengthy evaluation follows.

Advantages

We are starting the IxRetail effort with a data model already in hand. This approach leverages that model to further define a set of retail objects rather than a set of retail applications, which is both a more natural and less restrictive thing to do.

Here each IxRetail-compliant application signs a “contract” to supply certain of these objects and require certain others. The actual messages it sends are generic (object Create/Read/Update/Delete), so in a sense, except for the actual data objects involved, ALL IXRETAIL APPLICATIONS SHARE THE SAME INTERFACE.

Mapping such a model to legacy systems becomes significantly easier, as an existing application (typically front-ended by an “IxRetail Agent”) can be transformed to support an object-based contract, which reflects its own existing data usage needs. It no longer need first be revamped to exactly match the multi-document interface of one of the IxRetail application “types”.

Conversion of a legacy application to IxRetail-compliance now involves the following steps:

1. Determine the contract (IxRetail objects needed & supplied).

2. Create an “Agent” which connects the application to the IxRetail standard. Such an agent might interface with the application directly, or it might restrict itself to providing / updating information in the applications private database.

3. Supply the contracted objects supplied via the Agent

4. Arrange to get all needed retail objects via the same Agent.

In cases where a needed object supplier within the enterprise is not yet IxRetail compliant, continue to use the preexisting method for obtaining such object data, until such time as it becomes available via the Agent.

This option draws its strength from the assumption that the above guidelines are straightforward enough to ensure widespread adoption of the IxRetail standard.

Disadvantages

A. Reduced Functionality

This model reduces the functionality of the IxRetail standard. Instead of defining the complete document level interface for all applications in the retail enterprise, it "only" provides a "Retail Enterprise Data Sharing Facility" based upon the ARTS data model.

B. More Extensive Infrastructure

Application "types" are eliminated so that applications view the outside world as a single IxRetail Persistent Data Store. Since an application no longer knows the specific partners it needs to contact to exchange retail data, this model requires a centralized architecture, with each application communicating only with the central process which supports the data store.

It is this central process which maps each object request to the retail application that has contracted to provide it, and maps the response back to the requesting application.

The major disadvantage of this model is the requirement to deploy this central process, which is independent of all retail applications and yet must somehow be supplied (either by separate vendors or by open source donated to the IxRetail committee) as part of the infrastructure to enable the standard to be successfully deployed.

At its simplest, this process is a store & forward router for IxRetail XML messages. If we require a publish and subscribe facility (see below) things become more complex, as the central process must also support persistent event queues which guarantee later delivery of published events to subscribing applications, even if these events occurred while the application was offline.

Note that this model is very similar to the one used for the School Interoperability Framework (SIF), which unites all data within a school district in just this fashion. The required SIF central process is currently supplied free to all members for both the Java and Windows platforms by Sun Microsystems and a major NT ISV respectively. We might realistically expect a similar situation to occur for IxRetail.

Conclusions

The Alice model allows easy mapping of legacy applications to IxRetail at a cost of requiring deployment of an additional piece of "central" software for these applications to connect to.

In addition, while a given application will specify its object contract, a system administrator must still be involved to match producer and supplier contracts, and possibly do some level of integration work, before interoperability of IxRetail applications within a given retail enterprise can be achieved.

Thus while compliance requirements are greatly simplified, plug & play interoperability only remains possible between “allied suppliers”, but is not guaranteed in the general case.

Note that the decision of whether or not to base the IxRetail infrastructure on the Alice model will have great impact on many of the choices we face below. THIS IS A KEY DECISION.

4. Partner Discovery

Before two applications can communicate, they must first locate each other. There are several common techniques by which this may be accomplished.

A. Dynamic

Under this model, partner discovery is made dynamically at process startup or only when needed. This requires the accessibility of either an enterprise-wide Naming Service or Process Registry.

Normally the contents of such a registry are preset by the network administrator. Given the URI for the registry and a “tag” for each process, the actual partner process address (URL, email address, node / port #,...) can be found, and interprocess communication can begin.

.

B. Very Dynamic

In the extreme case, the registry may itself be dynamically populated by having the individual applications publish their personal APAs (Application Partner Agreements). If this option is selected, the new UDDI standard for dynamic partner registration and discovery might be quite useful in providing the necessary functionality.

C. Static

The location of each partner is hard-wired into the application (or its configuration data). There is no partner discovery, and neither an enterprise-wide Naming Service or Process Registry is required.

D. Very Static

In the extreme case, the static option maps directly to the “Alice” architecture, in which an application has no way to identify the processes it needs to interoperate with. Instead it uses the central process as a store and forward router, to:

1. Convert the destination object name in the message into the location of the process which services that object.

2. Route the message to that process

Conclusions

Option A

This uses a standard naming service to locate the various IxRetail applications. It would require us to define the “generic names” under which these applications could be found in the typical retail enterprise (ex: “GiftRegistry”). If any additional partner information was required, we would need to incorporate some sort of “process registry” into IxRetail, and define the APA information in detail as part of the IxRetail standard.

Examples of such data might include:

Application “type” and/or Objects supported
IxRetail version number
Optional XML fields supported
Push or Pull mode (see below)
Alternative transports supported (see below)

Option B:

It is hard to imagine a retail enterprise that would allow a newly installed application to dynamically register itself, discover its own partners, and then begin exchanging data. Aside from possible catastrophic effects from application incompatibilities, this scenario raises serious security concerns.

Option C

This also seems unacceptable, since every time a new application is installed, the configuration data for all other applications which communicate with it have to be altered, and any associated incompatibilities have to be resolved.

Option D

The other viable choice, and the only one that supports the “Alice Model”. As there is NO partner configuration data (only the names of the objects the application wishes to receive or supply), any configuration changes that occur will be within the central process (i.e. the infrastructure), and will not impact the existing applications when new applications are removed or installed.

5. One-Shot Messages vs. Sessions

This issue addresses the question of whether two communicating IxRetail processes can or must establish a “session” to support stateful parameters which simplify the messages they exchange.

One-Shot Message:

Here each message exchanged between two applications is self-contained. If security information is required, it must be fully verifiable. For example, a process might have to send an X.509 certificate in each message to its partner, which would then have to be revalidated each time.

Transaction support is impossible because there is no inter-process session to hold the necessary state information. Additional administrative procedures would be required to ensure interoperability (w.r.t. compatible IxRetail versions, quality of service, security, etc.).

On the other hand, since there is no interprocess state, the sudden loss of a partner can be easily handled, and the resulting system is more robust. This option is therefore particularly well suited for applications which are connected across the Internet, or which communicate rather infrequently.

Long Term Session:

This approach requires the creation of a session between any two communicating applications (or in the Alice model, between any application and the central process).   The session state variables may be dynamically negotiated during a “session creation” message exchange. Alternatively, IxRetail would have to allow the static definition of all session state information, via some administratively controlled Application Partner Agreement.

Typical “session state” information consists of some or all of the following:

A. Security Cookie

The following scenario assumes that the transport in use (ex: HTTP) does not support partner authentication, and that it therefore has to be supported at a higher layer of the infrastructure.

Here a session requester attaches an X.509 certificate to the first message it sends to its partner. The partner validates it (a rather time consuming process), which authenticates the requester. It then creates a cookie which it stores as part of its session state, and returns a copy to the requester.

Whenever the requester sends another messages, it includes this cookie. The partner can therefore authenticate all subsequent message by simply comparing the cookie each contains with the cookie saved in session state. For partners which exchange multiple messages in a secure environment, this represents a significant performance optimization over re-performing certificate verification for each arriving message, as would be required in the "One Shot" case.

B. Push/Pull Flag

This option determines the subsequent flow of data between two partners at the time the session is established. "Push" mode is standard... if one partner has a message for another, it sends it. As this arrives asynchronously at the receiver, it usually requires the receiver to support a multi-threaded "arrived message queue", with one thread popping messages off the front for processing, and another inserting newly arrived messages at the back.

Assuming the underlying transport was HTTP, supporting PUSH mode means supporting incoming HTTP connections, which means the partner cannot be based on a browser, and firewall considerations come into play.

Pull mode, limited to one partner (A) in a session, blocks its partner (B) from sending a message until it receives an IxRetail "GetNextMessage". As a result, partner (A) receives all incoming messages synchronously (i.e. only when specifically asked for). This is true even for events, which are normally asynchronous by nature.

As a result, (A) requires no special queueing for incoming messages. A side benefit, again assuming the underlying transport was HTTP, is that (A) can be running on a browser, since an application operating in Pull mode is the one that always initiates all HTTP connections (by sending the "GetNextMessage").

Note that in the Alice model, ALL the retail applications can operate in Pull mode since they communicate directly only with the central process.

C. IxRetail Version
Once established or negotiated at session creation, each partner knows the version of IxRetail it must conform to whenever it communicates with its partner. The only assumption is that subsequent versions of the session creation message (which establish the session) must be backwards compatible (so that say a version 2 partner can at least identify that what it has received is a version 1 session request message).

D. Alternative Infrastructure
Assuming mutual agreement among BOTH partners (dynamically during the initial exchange of the session creation message or statically via administrative procedures), an infrastructure other than the IxRetail reference might be used for the duration of the session (ex: MQ/Series).

E. Transaction State
This component is different than the others, since it is a variable rather than a constant. During the lifetime of the session, the Transaction State continually cycles through a set of values which reflect the “choreography” of various transactions the session partners are supporting.

There are several efforts now underway to standardize on XML “workflow” schemas that we could look to adopt if the IxRetail use cases require transaction support.

Conclusions

We may find as other standards have (HitisX, OTA) that both the “one-shot” and session options need to be supported.

6. Immediate vs. Delayed Responses

The choice made here will control how data will be exchanged between two IxRetail applications. For purposes of illustration, HTTP will be assumed to be the infrastructure transport layer in the analysis below, although other transports would raise similar issues.

Assume an application requests information from a partner and the partner responds with the data, thereby completing the message exchange. There are two ways in which this can take place.

A. Immediate Response
If HTTP is the transport, a single HTTP connection could support the entire Request / Response data interchange.

A. The "RequestData" function is issued. This opens an HTTP connection by sending the Request message to the partner via an HTTP POST.

B. The partner receives the Request, processes it, and issues a Response message. This is conveyed via an HTTP POST in the opposite direction, which closes the session.

C. The returned data is passed back to the issuer of the "RequestData" function, as part of the function return.

Advantages

The entire request/response sequence is supported within a single synchronous RequestData function call. This presents a straightforward API to the requested application.

Also, by using only a single HTTP connection, system performance is optimized.

Disadvantages

Depending upon the complexity of the actual request, and the bandwidth and latency of the connection between the two applications, the remote partner may not be able to respond within a reasonable timeframe. The requesting partner is then faced with the dilemma of whether the remote partner is actually working on the request, or whether some sort of network error occurred.

This option can therefore result in many open but inactive HTTP connections, and significant blockage to the requesting application. A fairly involved "timeout" analysis is often required, and the infrastructure may have to support an asynchronous "CancelRequest" message to allow the requesting process to unblock.

B. Delayed Response

Here the original Request message is "Ack'd" by the remote partner (via an HTTP POST) which closes the first connection, without any response information being sent.

Later, when the actual response information does become available, the remote partner opens up a new HTTP connection and sends this data. The requestor receives it, and issues an "Ack" message (via a final HTTP POST) which closes the second HTTP connection.

This requires the infrastructure to support a “RequestID” in each request, which is then returned in the response message. This allows the requesting process to correlate request/response pairs.

Advantages

Long delays between Request and Response do not leave open HTTP connections and do not block the requester.

Disadvantages

The programming model is more complex. Arrival of a response occurs asynchronously, similar to the arrival of an event. Additional code in the requester must place the response in an incoming message queue, a separate thread is required for each simultaneously active request and additional request / response matching logic must be present.

Conclusions

No real surprise here. Immediate response should be the choice, unless the use cases demand support for delayed response.

7. Generic vs. Specific Request Schema

The choice here is between defining a separate, specific Request Doctype for each type of response required, as:

```
<RequestItem>         ***    </RequestItem>

<RequestCustomer>  ***    </RequestCustomer>

<RequestGiftList>     ***    <RequestGiftList>
```

as opposed to defining a single GenericRequest Doctype capable of specifying the format of all possible responses. An example of this is shown below, which requests the description and price of all items costing less than $200.

```
<GenericRequest ObjectType = ITEM>
   <Fields>
      <Field> Description </Field>
      <Field> Price </Field>
   </Fields>

   <Conditions>
```

```
          <Condition>
             <Field> Price </Field>
             <Comparitor   type = LessThan   />
             <Value> 200.00 </Value>
          </Condition>
       </Conditions>
    </GenericRequest>
```

A. Generic Request: Advantages
This option is required for the Alice Model, where the central process routes the request (based only on ObjectType desired) to the appropriate application for servicing. So in effect the requester's view (of the central process and the responding application together) is of a central database, with the generic Request message corresponding to an SQL query on that database.

As a result, the requesting application is not restricted to asking one of a fixed set of partners to supply a fixed set of data structures. Rather it issues a generic Request message to the "shared database" to get only the data it needs.

However a Generic Request capability is also quite useful for alternative models as well, as it provides increased flexibility and significantly reduces the total number of XML Schemas required to define the standard.

B. Specific Requests: Advantages
The main advantage of having multiple "data specific" request messages is that this approach allows the XML parser to do much more w.r.t. message validation.

The analogy here is to the OO option, which provides a single object with a "type" member, vs. subclassing that object into a separate subobject per "type". The subobject approach allows the compiler to automatically validate that the collection of additional functions and data used, actually corresponds to the subclass declared.

The GenericRequest schema, with its element values for such things as object field name and field value, leaves much more of the subsequent validation (does this field name even exist for this object?, is this value legal for this field?, etc.) to the receiving application.

8. Publish and Subscribe Capability (Y/N)

Use cases may demand that one application possess the ability to “monitor” the data structures controlled by another, being asynchronously alerted when changes (new occurrence created, existing occurrence modified) are made.

The way this is usually achieved is via a Publish / Subscribe capability. The “monitor” process subscribes to the “data structure changed” events which the data owner promises to publish. An example is the “PriceChanged” or the more general “ItemChanged” event. Supporting such a capability imposes the following requirements on the IxRetail infrastructure.

1. Identification of a fixed set of data structures (or data objects) upon which events can be reported.

2. Definition of additional IxRetail message headers for the following:

   Subscribe and Unsubscribe
   If the subscriber only knows the object name for the data it wants to subscribe to, the infrastructure must map the subscribe request to the application which will publish the event.

   Publish and Unpublish
   If the publisher doesn’t keep track of subscribers, the infrastructure must multiplex each published event to the entire subscriber list.

   Event
   If the subscriber expects guaranteed event delivery (i.e. events occurring when it was offline are reported when it connects back online) and this is not supported by the publisher, it must be guaranteed by the IxRetail infrastructure.

Note: In the Alice Model, it is the central process (i.e. part of the IxRetail infrastructure) rather than the individual retail applications, that supplies all the above services.

A typical sequence utilizing a publish / subscribe capability might look something like the following:

X1 announces it publishes “change data” events for object Y.

X2 and X3 announce they each subscribe to “change data” events for object Y.

X1 creates a new object Y1. It publishes a change event to report this.

X2 and X3 receive this event. X2 ignores it.

X3 issues a request for the object which has changed.

X1 receives this request and issues a response which contains the complete data for this object.

X3 receives this response.

Note that this assumes the existence of a publisher-unique "object identifier" (or UID) which must be included in the event message and which, returned in the request, allows the publisher to identify the object being requested.

We would therefore need to define such a UID element in the IxRetail standard for all objects on which events were reported, or the infrastructure would have to require that the publisher include an entire copy of the object being changed (as opposed to just the UID) within the event message. Use cases, object sizes and bandwidth considerations will all be factors in making this design decision.

Finally, depending on security constraints, Access Control Lists (ACLs) might have to be employed to determine which applications within an enterprise were allowed to publish and/or subscribe to which objects. Typically such ACLs would be set up administratively, with only a "RoleId" field required in the Header layer to support it.

9. Manifests (Y/N)

Manifest functionality is typically supplied by the Envelope layer of the infrastructure rather than the Header, as it deals with packaging issues. As we are planing to "take" rather than "make" the Envelope layer, we can assume that the rather involved infrastructure needed to support manifests will be provided, and we need only concentrate on how to leverage the functionality they provide.

Basically a Manifest forms the basis of a "compound document" that supersedes a single XML message as the unit of data exchange between two IxRetail processes. This compound document consists of the following INDEPENDENT components:

One Compound Envelope Header (XML)

One Manifest, containing the description, name and version of each subsequent payload (XML)

Zero to N Payload documents (some of which may be XML)

A typical vertical XML standard such as IxRetail would normally define only the payloads, whereupon the Envelope layer generates the Manifest and Header, and wraps the whole thing into a single compound document.

The major advantage offered by the Manifest is the ability to completely separate the Header layer of the infrastructure from the rest of an IxRetail document.

The Problem

Assume we have defined a Response message, supported by XML elements in the Header layer. The problem is that the Response data can contain ANY collection of retail objects defined by ANY working subcommittee, all wrapped in the Response Header data defined by the Infrastructure subcommittee.

What we really need is the ability to strip off and process the “Response-Header” in one routine (via say a SAX parser) to ensure that the associated object data is meaningful, by checking if:

It is well formed XML.
It is a valid XML ResponseHeader.
Its source ID matches an outstanding request.
It arrived before its timeout expired.

Only after the ResponseHeader is validated is the context for the included Response data revealed. At that point we want to pass the remainder to a second, context-specific routine to validate (perhaps via a DOM parser) and process the rest of the Response message. The problem is that, due to the “magic” of XML, we can’t do this!

Consider a typical Response message (excluding transport and envelope).

```
<ResponseMessage>          // Doctype
    ...... Header Stuff ........
   <PriceData> ***   </PriceData>
</ResponseMessage>
```

Note again that due to the trailing “/ResponseMessage” tag, the infrastructure SURROUNDS the Price data rather than simply preceding it. Thus the Header cannot be separately validated, since it is joined at the hip to the pricing data elements (as well as to any other elements which might possibly be returned) within the ResponseMessage schema.

Some Potential Solutions

Four solutions exist which attempt to resolve the conflicts introduced by packing Header layer and retail object data into the same XML message. The first three approaches limit themselves to changes in the Header level. The fourth (Manifests) attacks the problem from underneath, by enhancing the Envelope layer.

A. A Single Humongous Response Schema:

Here the schemas for all possible retail objects that could be returned in a Response message are “OR’d” together inside the ResponseHeader schema to create one massive schema which completely defines the format of any/all IxRetail Response messages. The on-the-wire XML looks something like the above, where any other type of data element could be substituted for PriceData.

The advantage is that only one schema exists for ResponseMessage. Since this need not contain any optional elements, or elements for which the internals are unspecified, the level of automatic validation that can be provided by the XML parser is high.

The big problem is the instability such a huge schema definition introduces. Basically ANY change to ANY piece of data in ANY object causes a schema change for ResponseMessage, immediately outdating the schemas used by all IxRetail applications which send or receive ResponseMessages. This is true even for applications that are otherwise independent of the object which was changed. Version compatibility matching and schema maintenance issues get very ugly very quickly.

B. A Single Generic Response Schema:

In a manner closely paralleling the generic Request, this option creates a single generic ResponseMessage which, as it does not define the actual object data, is the same for all Responses.

```
<ResponseMessage>       // Doctype
```

```
        ...... Header Stuff ........
       <ResponseData  type = PRICE>
                  .....                  // Undefined Response data internals!
      </ResponseData>
    </ResponseMessage>
```

The problem is identical to that of the generic Request option, namely an inability to automatically verify the contents of the message. Only here it is MUCH worse.

Consider that while the ResponseData element may have an enumerated list of types (one per data object) its internal contents must be undefined in the schema. Thus every IxRetail application would have to include code to verify that each and every XML element it expected was actually present in the data of each and every ResponseMessage it received. Ugh.

C. Multiple "Narrow-Focused" Response Schemas:

This option requires us to define a completely separate XML schema for each type of data that could possibly be returned in a Response message. Note the large number of new document types that result:

```
<PriceResponse>                    // Doctype
    <ResponseHeader>
       ...... Header Stuff ........      // IDENTICAL for all Responses
    </ResponseHeader>

    <Price>
        ***
    </Price>
</PriceResponse>
```

This is close to being an acceptable solution, although it still suffers from the following drawbacks:

A. Additional Schemas
The large number of additional Doctypes require the same number of additional XML schemas to be added to the standard. This causes a growth in standard "size" and perceived complexity.

B. Contested Ownership
There is no clear ownership for these new schemas, as they consist of

component elements supplied by both the individual subsystem workgroups and the infrastructure committee. Who owns maintenance?

C. Versioning Conflicts
Whenever the work products of two disjoint groups get lumped together in a single message, versioning conflicts result.

If the Price data changes, the version of the PriceResponse should be updated. If the ResponseHeader data changes, the version of ALL Response messages should be updated. So what does the version number of the PriceResponse really reflect?

D. Manifests:

None of the above solutions would permit the IxRetail developer to separately validate the message Header and Data components, which was the original problem. If we use the Manifest capability of the Envelope layer to define separate payloads for the Header layer and the object data in the Response message, the compound document looks like the following:

Manifest
```
<Manifest>
   <Payload Name = ResponseHeader Version = 2.0 />
   <Payload Name = ItemResponse Version = 3.2 />
</Manifest>
```

Payload 1
```
<ResponseHeadert>
   <SourceID>     aaa </SourceID>
   <DestinationD> bbb </DestinationID>
   <RequestID>    ccc </RequestID>
   <DateTime>     ddd </DateTime>
</ResponseHeader>
```

Payload 2
```
<ItemResponse>
   xxx          // Pricing Data
</ItemResponse>
```

Advantages

Modularity
Each Payload can be separately (and completely) verified via DOM by independent software modules.

Such modularity also allows us the possibility of replacing the entire Header layer of the infrastructure by an alternative choice, without affecting any of the existing XML schemas generated by the other workgroups (or affecting the retail logic of existing IxRetail-conformant applications).

Versioning
Each Payload has its own version number specified in the Manifest, so its schema can be modified and published by independent working groups.

Additionally, since the Payload version # is now external to the actual payload data itself, an application does not have to parse an XML Payload to determine if it is conforms to a version of IxRetail beyond what it can handle.

Manifests therefore have the potential for greatly simplifying the versioning problems we otherwise would face in the future. (Note that ANY versioning scheme we might chose would probably work perfectly for version 1)!

Binary Attachments
Some of the Payloads can contain non-XML data. This might be particularly useful if an application needed to return say a JPEG file to represent the catalogue image as part of the response to an item lookup.

Conclusions

This document identified a fairly wide set of possible options for the design of the IxRetail infrastructure. We first need to review and extend this list.

After that ... **Bring on the use cases!!**