

The Highs and Lows of Integrating LDAP with XML

Steven Legg

Abstract

LDAP is commonly used as a repository for data about objects, particularly users, of relevance to the operation of an enterprise's computing resources and applications. However, there is a strong tendency for newer application protocols and their data payloads to be defined in an XML schema language for rendition as XML documents. Such applications typically have a need for the kind of user and other object data that has traditionally been held in the entries of an LDAP directory service, as well as defining new kinds of structured data to be associated with those same objects. There is a clear advantage to data consistency and ease of administration in having a common directory service for all of an enterprise's applications, whether or not they are based on XML. Unfortunately, the core LDAP specifications have no inherent support for XML-formatted object data or XML-formatted protocol messages. This paper identifies four approaches to marrying XML with directory services and discusses the advantages and disadvantages of each approach with respect to simplicity, utility, uniformity and extensibility.

The first approach encompasses ad-hoc solutions using the existing LDAP framework. For example, embedding XML documents in directory attributes with a string syntax, or defining specific-purpose syntaxes and matching rules.

The second approach includes LDAP-inspired XML-based directory protocols such as the Directory Services Markup Language (DSML) version 2 and the Service Provisioning Markup Language (SPML), which may be readily implemented as alternative interfaces to the directory service.

The third approach involves XML-based registry or discovery services, as exemplified by Universal Description, Discovery and Integration (UDDI) and the ebXML Registry, which bear no particular relationship to LDAP, but provide a similar function and

can be considered to be competing against LDAP.

The fourth approach is the XML-Enabled Directory (XED) framework, which is a collection of extensions to ASN.1, LDAP and X.500 designed to seamlessly support XML within the LDAP/X.500 directory model, for both data and protocol. This paper shows how XED achieves most of the advantages of the other approaches while avoiding their more serious drawbacks.

1 Introduction

LDAP [4] is commonly used as a repository for data about objects, particularly users, of relevance to the operation of an enterprise's computing resources and applications. However, there is a strong tendency for newer application protocols and their data payloads to be defined in an XML schema language for rendition as XML [19] documents. Such applications typically have a need for the kind of user and other object data that has traditionally been held in the entries of an LDAP directory service, as well as defining new kinds of structured data to be associated with those same objects, for example, eXtensible Access Control Markup Language (XACML) policy sets [25]. There is a clear advantage to data consistency and ease of administration in having a common directory service for all of an enterprise's applications, whether or not they are based on XML. Unfortunately, the core LDAP specifications have no inherent support for XML-formatted object data or XML-formatted protocol messages. This paper considers four general approaches to overcoming this deficit.

The first approach encompasses ad-hoc solutions using the existing LDAP framework. This basically entails embedding XML documents in directory attributes with either the Octet String or Directory String attribute syntax [5].

The second approach includes LDAP-inspired XML-based directory protocols, which may be read-

ily implemented as alternative interfaces to an LDAP directory service. Two examples are considered in this paper: the Directory Services Markup Language v2.0 (DSMLv2) [23] and the DSMLv2 Profile [29] of the Service Provisioning Markup Language Version 2 (SPMLv2) [28]. The DSMLv2 Profile of SPMLv2 adopts the DSMLv2 representation for directory attributes, so the comments made with respect to the characteristics of DSMLv2 generally also apply to this profile of SPMLv2.

The third approach involves XML-based registry or discovery services, which bear no particular relationship to LDAP, but provide a similar function and can be considered to be competing against LDAP. Three examples are considered in this paper: Universal Description, Discovery and Integration (UDDI) [24], the ebXML Registry [26] and the XSD Profile [30] of SPMLv2 (which unlike the DSMLv2 Profile of SPMLv2, has very little in common with LDAP).

The fourth approach is the XML-Enabled Directory (XED) framework [7]; a collection of extensions to ASN.1 [11], LDAP and X.500 [9] designed to seamlessly support XML within the directory model that underlies both LDAP and X.500.

The XED framework defines a number of protocols. Of particular interest here is the XML Lightweight Directory Access Protocol (XLDAP) [8], which is a rendition of LDAP as a pure XML-based protocol. Although XLDAP is more closely related to LDAP than is DSMLv2, it is sufficiently different in character to DSMLv2 that it warrants being considered as a separate approach.

XED also provides XML renditions of each of the X.500 protocols, but these are not explicitly considered here since they have the same essential characteristics as XLDAP. Common to all these XED protocols is the substitution of the Basic Encoding Rules (BER) [12] of ASN.1 with a new set of ASN.1 encoding rules, called the Robust XML Encoding Rules (RXER) [6], that produce XML as the encoding. In the terminology of ASN.1, this makes XML a transfer syntax.

This paper examines the advantages and disadvantages of the representative protocols of each approach with particular emphasis on XLDAP, which has been deliberately designed to leverage the advantages of the other approaches while avoiding their disadvantages. Note that many of the capabilities of XED are accessible through LDAP, but comparisons to LDAP in this paper assume an LDAP server without these capabilities.

Section 2 compares the various protocols with respect to their consistency in the use of XML as a transfer syntax.

Section 3 examines particular problems associated with the use of XML as the transfer syntax for protocol messages and how these problems affect the protocols being considered.

Section 4 compares each protocol's ability to perform queries over XML-formatted data.

Section 5 examines the extensibility of the various protocols from the point of view of end-users.

The advantages of each protocol are summarized in section 6.

2 Uniformity

This section compares the various protocols from the point of view of syntactic uniformity. Section 2.1 considers the variety of different transfer syntaxes employed by each protocol, and section 2.2 examines discontinuities in the use of XML for data payloads.

2.1 Single Transfer Syntax

Each of the LDAP requests and responses has a corresponding XML representation in DSMLv2. Directory attribute values appear in DSMLv2 messages as the content of nested `<dsml:value>` elements. As an example, the object class description for the person object class (a value of the `objectClasses` directory attribute) would appear in a search result for DSMLv2 like so (as a single line):

```
<dsml:value>( 2.5.6.6 NAME 'person' SUP top
STRUCTURAL MUST ( sn $ cn ) MAY ( userPasswo
rd $ telephoneNumber $ seeAlso $ description
) )</dsml:value>
```

The "dsml:" namespace prefix is assumed to be defined on one of the elements in the DSMLv2 message that encloses the `<dsml:value>` element.

Obtaining the component parts of this object class description requires the use of more than one kind of parser in a DSMLv2 implementation. An XML processor (i.e., a parser for XML) parses the protocol message containing the `<dsml:value>` element to obtain the character string value of the directory attribute value, and a parser for the LDAP-specific encoding [5] of the Object Class Description syntax must be invoked to identify the various component parts of the object class description.

An implementation of the DSMLv2 Profile of SPMLv2 also requires a parser for the LDAP-specific encodings of various LDAP syntaxes since it directly reuses the directory attribute value representation of DSMLv2.

A DSMLv2 implementation further requires a BER encoder and decoder to process LDAP controls

and extended operations, which are represented in DSMLv2 as the base64 encoding [1] of a BER encoding (controls and extended operations are not used by the DSMLv2 Profile of SPMLv2).

For the sake of parsing simplicity, it is preferable to have a single transfer syntax for a protocol message and its data payload, and therefore the need for only one kind of parser.

The protocols of X.500 have the property of requiring only a single transfer syntax (i.e., BER) for both protocol message and payload. This property was lost in the design of LDAP, where directory attribute values are conveyed as the content of ASN.1 OCTET STRINGS. XLDAP reverses the process by replacing LDAP's OCTET STRING containers for directory attribute values with the open types used by X.500, with the result that an XLDAP message and the directory attribute values that it contains are all part of one continuous RXER encoding. Returning to the example, the object class description for the person object class would appear in a search result for XLDAP as the following markup (the `<value>` elements in XLDAP are not namespace qualified):

```
<value>
  <identifier>2.5.6.6</identifier>
  <name>
    <item>person</item>
  </name>
  <information>
    <kind>structural</kind>
    <mandatories>
      <item>2.5.4.4</item>
      <item>2.5.4.3</item>
    </mandatories>
    <optionals>
      <item>2.5.4.35</item>
      <item>2.5.4.20</item>
      <item>2.5.4.34</item>
      <item>2.5.4.13</item>
    </optionals>
  </information>
</value>
```

Although they use different data payloads, the ebXML Registry, UDDI and the XSD Profile of SPMLv2 share the property of XLDAP of using a single transfer syntax for both protocol message and payload. Note that this is not an absolute assessment in that there may be payloads containing DER encodings that, for pragmatic reasons, need to be preserved (e.g., X.509 certificates [10]). This need will affect all the protocols equally and so is not a distinguishing feature. An implementer will likely be able to pass such encodings to a security API without having to invoke an additional decoder.

2.2 Continuous Encoding

There is a prevailing bias that values of the human-readable syntaxes of LDAP are “just” character strings. While this is true of the most commonly used LDAP syntaxes, there are numerous examples (more so in X.500) of syntaxes that represent structured data. One example is the Object Class Description syntax used in the previous section. It is more in the spirit of XML for structured data to be represented as element content rather as character data. The expectation that directory attribute values are only character data causes further awkwardness in DSMLv2.

Suppose that a user wants to store, as the value of a new directory attribute, an XML-formatted inventory record like the following:

```
<product>
  <code>XJ459</code>
  <quantity>137</quantity>
</product>
```

If we assume that this inventory record is stored in a directory attribute with the Directory String syntax, then the value of the attribute would appear something like this in a search result for DSMLv2:

```
<dsml:value>
  &lt;product>
    &lt;code>XJ459&lt;/code>
    &lt;quantity>137&lt;/quantity>
  &lt;/product>
</dsml:value>
```

DSMLv2 allows the content of the `<dsml:value>` element to be a URL pointing to a location containing the octets of the directory attribute value. Such a pointer is inconvenient for a receiver to process and won't be considered further here. In the typical case where the directory attribute values appear in-line, the content of the `<dsml:value>` element is required to be character data conforming to either the XML Schema [17] [18] string type or the XML Schema base64Binary type. DSMLv2 isn't specific about the circumstances under which the string or base64binary types are used, but it is likely that the string alternative is intended for directory attribute values with a human-readable syntax (e.g., the Directory String syntax, as in the preceding example), and the base64binary alternative is intended for the rest. A consequence of conformance to the XML Schema string type is that element content (i.e., content with child elements) is not permitted, and any character that has special meaning to an XML processor has to be escaped. There are various ways of escaping the special characters in the

directory attribute value (e.g., a CDATA section [19] could be used), but the effect is the same whatever escaping mechanism is used; a conformant XML processor returns a simple character string to the higher-level application.

If we assume that the XML document is stored in a directory attribute with the Octet String syntax, then the value would instead appear something like this (as a single line):

```
<dsml:value xsi:type="xsd:base64Binary">PHB
yb2R1Y3Q+DQogPGNvZGU+WEoONTk8L2NvZGU+DQogPH
F1YW50aXR5PjEzNzwwcXVhbnRpdHk+DQo8L3Byb2R1Y
3Q+DQo=</dsml:value>
```

The content of the `<dsml:value>` element is the base64 encoding [1] of the octets of the directory attribute value. The “xsi:” and “xsd:” namespace prefixes are assumed to be defined on an enclosing element. In this case, a conformant XML processor returns the string of characters in the base64 encoding to the higher-level application.

Whether the Directory String or Octet String syntax is used for the directory attribute, obtaining the component parts of the inventory record requires a second invocation of an XML processor (after a base64 decoding in the latter example) on the character data obtained from the first invocation of the XML processor. This need for recursive applications of the XML processor is undesirable from the point of view of ease of implementation, although there are some advantages, which will become apparent in the next section.

Using the capabilities defined in XED, it is possible to define a directory attribute to hold the XML-formatted inventory records such that they appear in their natural form in XLDAP (i.e., without escaping). The value of the directory attribute value would appear like this in a search result for XLDAP:

```
<value>
<product>
  <code>XJ459</code>
  <quantity>137</quantity>
</product>
</value>
```

So in XLDAP, the protocol operation and the element content data payloads are able to be processed in a single pass of an XML processor, i.e., XLDAP has a continuous encoding.

A continuous encoding is achieved in the XED framework by defining a special ASN.1 type, called the Markup type [6], whose abstract values can hold the content and XML attributes of an XML element,

and which is analogous to the XML Schema any-`Type`. Essentially, the content and attributes are embedded in instances of the ASN.1 UTF8String type, but RXER recognizes the special nature of this type and an RXER encoder knows to output the content of Markup values in-line.

By default, directory attributes with the Markup type as their syntax use RXER as their LDAP-specific encoding. This is assumed to be the case with the inventory record example, so the preceding XML also shows how the directory attribute value will appear as the content octets of an LDAP attribute value, i.e., with an apparently extraneous `<value>` element enclosing the `<product>` element. This arises because a directory syntax is defined by a data type, which defines only the content and XML attributes of an element, rather than by an element definition. The enclosing `<value>` element packages the content and XML attributes into a complete element so that the LDAP-specific encoding is always a well-formed XML document.

Although they use different data payloads, the ebXML Registry, UDDI and the XSD Profile of SPMLv2 share the property of XLDAP of having a continuous encoding (notwithstanding the special case for DER encodings mentioned in the previous section). Like DSMLv2, the DSMLv2 Profile of SPMLv2 requires element content to be escaped.

Although having a uniform transfer syntax is a clean and elegant solution, it does have problems when that transfer syntax is XML. This is discussed in the next section.

3 Scoping Issues

The BER encoding of an abstract value has a regular structure wherein the octets for a nested abstract value are themselves a well-defined and complete BER encoding. This means that, at the syntactic level, it is possible to extract the encoding of a nested abstract value and perform operations on it (e.g., store, index, search and retrieve) without regard for the larger encoding from which it was extracted. This is also true for the GSER [2] encodings used by some newer LDAP syntaxes; the octets for a nested abstract value in a GSER encoding are a well-defined and complete GSER encoding. There may well be a semantic connection between the nested abstract values in an encoding, but for the purposes of this discussion we can assume that the higher-level application design maintains the necessary connections. For example, in LDAP and X.500, knowledge of a directory attribute type is required to interpret a directory attribute value. The directory protocols

are designed so that wherever an attribute value appears, it is associated with an attribute type that also appears.

The basic building block of an XML document is the element structure. Unfortunately, it is not generally true of an XML document that any nested element is a well defined and complete XML document when considered in isolation. This is because a nested element may contain qualified names [20] with namespace prefixes that depend on namespace declarations in an enclosing element, or may contain entity names [19] that depend on the Document Type Definition (DTD) of the enclosing XML document.

Issues surrounding qualified names and entity names are examined in sections 3.1 and 3.2, respectively.

3.1 Namespace Accumulation

Consider this short XML document, representing an operation of a simple, fictitious protocol:

```
<p:addObjectRequest
  xmlns:p="http://protocol.example.com"
  xmlns:u="http://user.example.com">
  <p:object class="1">u:data</p:object>
</p:addObjectRequest>
```

In this example operation, the name of the `<p:object>` element is a qualified name that depends on the namespace declaration for the “p” namespace prefix in the start tag of the parent `<p:addObjectRequest>` element. A namespace declaration is inherited by descendant elements insofar as it is not overridden by another declaration for the same namespace prefix. The in-scope namespaces [20] for an element are the namespace declarations defined on the element itself, plus the (not overridden) declarations inherited from its ancestor elements. If an element that is part of the data payload (e.g., `<p:object>`) is extracted from an XML document for a protocol operation (e.g., `<p:addObjectRequest>`) in order to perform some application-specific task, then it is necessary to retain all of the namespace declarations in the in-scope namespaces that define namespaces used by qualified names within the element so that the element can continue to be interpreted correctly. It is sufficient for the purposes of this discussion to assume that the necessary namespace declarations are retained by being copied to the start tag of the extracted element.

It is always possible to recognize qualified names as the names of elements and XML attributes, but

this is not the only place where qualified names can appear. Consider the character data content of the `<p:object>` element, i.e., “u:data”. Although this content looks like a qualified name, it could just be a literal piece of text with a coincidental resemblance to a qualified name. In order to decide what is, or is not, a qualified name in the character data of an element (or the value of an XML attribute) it is necessary to know the data type of the element (or XML attribute). Even if a data type definition is available, there is a further complication. Elements and XML attributes can contain formatted character strings, such as XPath expressions [21], that contain qualified names, but have a format that is beyond the capabilities of the commonly used schema languages for XML to describe. The data type for such formatted character strings is often defined to be the XML Schema string data type, which reveals nothing about their true nature. So it is not only necessary to be able to deal with elements and XML attributes of unspecified data type, but also to be able to deal with elements and XML attributes where the data type is specified, but inherently unrepresentative. Given that it isn’t unusual or illegal for the syntax of a directory attribute to be unknown, and given that the specified data type is unreliable anyway, the safest approach is to ignore the specified data type when dealing with namespaces in XML.

Based on XML syntax alone, it is possible to examine the content of an element and its XML attributes to identify each instance of character data that *could* be a qualified name and to consequently determine the maximal set of potentially significant namespace declarations. This could be quite time-consuming in a large XML document. A much quicker and simpler strategy is to just assume that all the in-scope namespaces are significant.

By way of example, suppose that the server holding the object returns the following protocol message in response to a request to fetch the object:

```
<q:fetchObjectResponse
  xmlns:q="http://protocol.example.com">
  <p:object
    xmlns:p="http://protocol.example.com"
    xmlns:u="http://user.example.com"
    class="1">u:data</p:object>
</q:fetchObjectResponse>
```

The server arbitrarily chooses the namespace prefix, “q”, to bind to the namespace for the response message. If the receiver extracts the `<p:object>` element from the protocol message and takes the simple approach of retaining all the in-scope namespace declarations, then the extracted object becomes:

```

<p:object
  xmlns:p="http://protocol.example.com"
  xmlns:u="http://user.example.com"
  xmlns:q="http://protocol.example.com">
  class="1">u:data</p:object>

```

The receiver's version of the element has an additional namespace declaration compared to the sender's version of the element. In this case, the receiver could have avoided adding the namespace declaration for the "q" prefix by checking whether this namespace prefix *appears* to be used within the element. Although such checking will minimize the addition of unnecessary namespace declarations, the possibility cannot be entirely eliminated. Consider what would have happened if the character data content of the `<p:object>` element had been "q:data". Assuming correctness, this character data cannot be a qualified name in the original `<p:object>` element since there isn't a namespace declaration for the "q" prefix. In arbitrarily choosing the namespace prefix "q", the server would inadvertently make the character data appear to be a qualified name. So in the general case, each time a data payload is composed into a new protocol operation by a sender and extracted by the receiver, the payload has the potential to acquire additional, unnecessary namespace declarations.

Note that namespace accumulation is not a problem for LDAP, DSMLv2 and the DSMLv2 Profile of SPMLv2 because any directory attribute value that is a complete XML document is transported as literal character data to be parsed in isolation by a separate invocation of an XML processor. The in-scope namespaces of a `<dsml:value>` element have no bearing on this independent interpretation of such directory attribute values.

The accumulation of unnecessary namespace declarations is an undesirable overhead in normal circumstances, but becomes a critical problem if the payload is subject to a digital signature.

Digital signatures are usually computed over a canonical representation of the data to be signed. Three well-known canonicalizations for XML are Canonical XML [14], Exclusive XML Canonicalization [15] and Schema Centric Canonicalization [27].

Adding any namespace declaration that was not in the original in-scope namespaces for the payload will change the Canonical XML representation of the payload (or some part thereof), and therefore break any digital signature computed using this canonicalization.

In contrast, the Exclusive XML Canonicalization of the payload (or some part thereof) is not changed by adding namespace declarations. Exclusive XML

Canonicalization is able to achieve this by augmenting the original payload with an InclusiveNamespaces PrefixList parameter, which is a list of the significant namespace prefixes. Any added namespace declarations will be omitted from the canonical form because their prefixes will not be in the list. However, Exclusive XML Canonicalization does not offer a general way to overcome namespace accumulation on directory attribute values because support for Exclusive XML Canonicalization has to be built into the data type definition for the payload, and only the creator of the payload knows for certain which namespace prefixes are significant. In other words, Exclusive XML Canonicalization cannot be reliably applied to a payload that does not already intrinsically support it.

Canonical XML and Exclusive XML Canonicalization both operate without regard for the underlying data type of the elements in an XML document. Schema Centric Canonicalization, on the other hand, uses knowledge of the data types to normalize the content and XML attributes of elements, and to replace namespace declarations and the namespace prefixes in qualified names. In the canonical form, each namespace that is used is declared as needed using a deterministically generated namespace prefix. The new namespace prefix is then used for all qualified names bound to that namespace within the scope of the declaration. Since the namespace declarations are reconstructed in the canonical form, the addition of unnecessary namespace declarations by a sender does not affect the canonical form generated by a receiver. As was noted earlier, relying on knowledge of the data type is problematic. The Schema Centric canonicalized form cannot be produced if the data type for an element is unspecified, but this might not be a serious problem in practice as it is only the producers and consumers of digital signatures that need to know the data type. Any number of intermediate systems, with or without knowledge of the data type, can handle the element without affecting the eventual consumer's ability to generate the canonical form. Schema Centric Canonicalization has special support for data types that are unrepresentative. In the terminology of Schema Centric Canonicalization, an XPath expression in character data is an example of an embedded language. To function correctly, Schema Centric Canonicalization requires that implementations understand each kind of embedded language that an application uses so that qualified names can be located and normalized. Each embedded language requires an identifier that is used to annotate the XML Schema definitions that drive the Schema Centric Canonicaliza-

tion. Since there is no limit to the number of different embedded languages that could be defined, the specification and implementation of Schema Centric Canonicalization is essentially open-ended. Obviously, Schema Centric Canonicalization will fail if a consumer does not understand an embedded language used by the producer.

Schema Centric Canonicalization could have been used in XED to minimize namespace accumulation on directory attribute values by using the annotations on imported XML Schemas to identify all the qualified names in character data, and therefore all the significant namespace declarations. However, the effectiveness of such a strategy would only be as good as the thoroughness with which imported XML Schemas are annotated, and the extent of the directory implementation's coverage of embedded languages.

XED takes a simpler approach that is compatible with the three canonicalization schemes for XML considered in this paper, and serves to limit the accumulation of unnecessary namespace declarations regardless of whether canonicalization is a consideration.

Observe that the namespace accumulation problem would go away if every element in an XML document carried all the namespace declarations it needed, and did not depend on any namespace declarations on ancestor elements. Any element could then be extracted and composed into a new protocol message without alteration. The obvious problem with doing this is that it creates a proliferation of namespace declarations, since elements would be duplicating namespace declarations that they could otherwise be inheriting from their ancestor elements. The XED framework makes a compromise in that it requires only *some* elements to be self-contained with respect to namespace declarations, specifically, elements where the data type is the ASN.1 Markup type. In an ASN.1 specification, any element where the actual data type is not an ASN.1 type is represented in ASN.1 using the Markup type, with a reference to the non-ASN.1 data type definition, if one exists. Character strings that contain an embedded language have an actual data type that cannot be adequately expressed in ASN.1 (or XML Schema), so these strings are also represented in ASN.1 using the Markup type.

The requirement for self-containment means that any element with the Markup type as its ASN.1 type can be freely extracted and composed into new protocol operations without revising its namespace declarations. Obviously, such an element must be self-contained when it is first created by an XLDAP

client. However, note that this must also be the case for an LDAP client when directory attributes with the Directory String or Octet String syntax are used to store XML-formatted data. In order for the values of such attributes to be well-defined XML documents, they must be self-contained with respect to namespace declarations. In typical usage, the source data for values of these attributes will be complete, external XML documents, so self-containment is trivially satisfied. If the source data is an element *nested* in some external XML document, then it is necessary to identify the namespace prefixes used within the source data and provide the necessary namespace declarations from the in-scope namespaces. A client must do this regardless of whether it is using LDAP or XLDAP. Building the inclusive list of namespace prefixes for Exclusive XML Canonicalization is much the same problem.

Directory attribute values that have an ASN.1 type other than the Markup type as their syntax would normally follow the conventional ASN.1 strategy of being decoded into abstract values (by an RXER decoder). Only abstract values of the Markup type store namespace declarations and namespace prefixes (where self-containment applies). Abstract values of all the other ASN.1 types have no indeterminate dependencies on namespace prefixes since they have no embedded languages or qualified names. Consequently namespace declarations do not need to be (and in any case, cannot be) preserved beyond the decoding step for these abstract values. When these abstract values are subsequently output by an RXER encoder, namespace declarations are added as required. The overall effect of repeated decoding and encoding of directory attribute values in XLDAP will be the elimination of unnecessary namespace declarations. Unlike Schema Centric Canonicalization, RXER achieves this without the need to annotate data type definitions with information about embedded languages.

As mentioned earlier, a directory client or server might not know the syntax for a directory attribute it receives. This means that it is possible for an XLDAP client or server to receive a directory attribute value and not know whether the value is supposed to be self-contained. If the value is meant to be self-contained, then it is unwise to be adding any namespace declarations. If the value is not meant to be self-contained, then it is harmful to neglect adding a necessary namespace declaration. RXER provides a special XML attribute, called the context attribute, to address this situation. If a receiver does not know the ASN.1 type for an element, and therefore does not know whether self-containment applies,

then it adds sufficient namespace declarations from the in-scope namespaces to define namespaces used by qualified names within the element. In the simplest implementation, the receiver can just add all the in-scope namespaces. The receiver also adds a context attribute, the value of which is a list of the namespace prefixes of all the namespace declarations that the receiver has just added to the element. If the element already has a context attribute, then this indicates that the sender also does not know the ASN.1 type, and the receiver does not need to do anything further to the element (the sender has already added enough namespace declarations to define namespaces used by qualified names within the element). If a receiver knows that the ASN.1 type for an element is not the Markup type, then the receiver processes the element according to the type, and ignores the context attribute, if it is present. If a receiver knows that the ASN.1 type for an element is the Markup type and the element carries a context attribute, then the receiver strips away the context attribute and any namespace declaration that matches a namespace prefix in the value of the context attribute. This returns the element to its original state. If a receiver knows that the ASN.1 type for an element is the Markup type and the element does not carry a context attribute, then the element must still be in its original state.

The context attribute is a special feature of RXER encodings that does not explicitly appear in ASN.1 type definitions. That is, it is possible to use the context attribute on any element in an RXER encoding regardless of its ASN.1 type. Contrast this with Exclusive XML Canonicalization and Schema Centric Canonicalization which need to be supported by the underlying XML Schema definitions.

The approach that XLDAP takes to managing namespace accumulation is compatible with payloads that depend on Canonical XML, Exclusive XML Canonicalization or Schema Centric Canonicalization. Elements with the Markup type as their data type will either not have any namespace declarations added, or, by virtue of the context attribute, will have any added namespace declarations removed when the data type is discovered to be the Markup type. The net effect is that there will be no additional namespace declarations. This means that the Canonical XML representation of the payload will not be disrupted, although it is necessary to extract the payload from the protocol message before generating the Canonical XML representation. A payload that depends on the Exclusive XML canonical form or the Schema Centric canonical form would not, in any case, be disrupted by additional namespace

declarations, and these forms can be generated from the payload either in-situ or after extraction from the protocol message.

LDAP, DSMLv2 and the DSMLv2 Profile of SPMLv2 are also compatible with payloads that depend on Canonical XML, Exclusive XML Canonicalization or Schema Centric Canonicalization because they are not affected by the namespace accumulation problem. UDDI implementations are not guaranteed to support payloads that depend on Canonical XML or Exclusive XML Canonicalization. The ebXML Registry and the XSD Profile of SPMLv2 are susceptible to namespace accumulation and so cannot support payloads that depend on Canonical XML.

3.2 Entity Names

A nested element is not a well defined and complete XML document when considered in isolation if it contains entity names. Entities are declared and named in the DTD of an XML document and are categorized as either parsed or unparsed.

Consider another example operation of the fictitious protocol:

```
<!DOCTYPE p:addObjectRequest [
<!ENTITY externalData "2">
<!NOTATION JPEG
  "http://user.example.com/viewer">
<!ENTITY picture
  SYSTEM "http://user.example.com/pic.jpeg"
  NDATA JPEG>
]>
<p:addObjectRequest
  xmlns:p="http://protocol.example.com">
  <p:object class="#externalData;"
    data="picture"/>
</p:addObjectRequest>
```

In this example, the value of the `class` attribute of the `<p:object>` element is a reference to a parsed entity with the name “externalData”. This entity is declared on the second line of the XML document. An XML processor is expected to replace a reference to a parsed entity with the replacement text specified in the entity’s declaration. Thus the dependency between the data payload (i.e., the `<p:object>` element) and the DTD of the enclosing XML document can be removed by processing references to parsed entities *before* extracting the data payload. The canonicalization schemes for XML also require that this processing takes place.

It is always possible to recognize a reference to a parsed entity, but it is not always possible to recognize a reference to an unparsed entity. The entity declaration with the name “picture” is an example of

an unparsed entity. Consider the value of the `data` XML attribute of the `<p:object>` element, i.e., the character string “picture”. Although this looks like a reference to the picture unparsed entity, the resemblance could be coincidental. In order to decide what is, or is not, an entity name as the value of an XML attribute (or the character data of an element) it is necessary to know the data type of the XML attribute (or element), specifically, whether it is the XML Schema ENTITY type.

Similar to the way namespace declarations can be handled, it is possible to examine the content of an element and its XML attributes to identify each instance of a character string that *could* be the name of an unparsed entity and to consequently determine the maximal set of potentially significant unparsed entity declarations that need to be retained when a data payload is extracted. The simpler strategy is to avoid checking by assuming that all the unparsed entity declarations are significant. Either way, problems arise when it comes to composing new protocol messages containing the data payload because of one important difference between entity declarations and namespace declarations; namespace declarations can appear on any element, whereas entity declarations can only appear in the DTD, which is at, or is referenced from, the beginning of an XML document. If a data payload contains the name of an unparsed entity, then it is necessary for the DTD to provide an entity declaration for that name, and the complete set of unparsed entity declarations will not be known (and the DTD of the XML document cannot be finalized) until every data payload to be carried in a protocol message has been examined. Therefore, forming the protocol message either requires two passes over the intended content of the protocol message (one pass to find the entities and one pass to encode), or requires the bulk of the protocol message to be stored until the DTD is finalized. While either strategy would not be a serious problem for XLDAP messages, which are all relatively small, there would be a noticeable overhead when constructing large search results in X-DAP-IP or total refresh operations in X-DISP-IP (the XED framework’s XML-ized versions of X.500 DAP and DISP, respectively). There is also the problem that two different data payloads can use the same name for distinctly different unparsed entities. Resolving such a name clash requires renaming one of the entities, which in turn requires complete knowledge of the data types for the payloads.

These various problems could have been overcome by defining another special XML attribute in RXER that could appear on any element and which con-

tained any retained unparsed entity declarations. In this way, data payloads could be made self-contained with respect to unparsed entities as they are for namespaces. On the other hand, the SOAP messaging framework [13], which is commonly used to transport XML-formatted protocol messages (and can be used by all of the protocols mentioned in this paper except LDAP), does not allow a Document Type Declaration, effectively making the ENTITY data type unsupported. Given all the problems surrounding the ENTITY data type, and its general disuse, XED takes the easy approach of just prohibiting it.

Note that LDAP, DSMLv2 and the DSMLv2 Profile of SPMLv2 are able to support references to unparsed entities in data payloads because any directory attribute value that is a complete XML document is transported as literal character data to be parsed in isolation by a separate invocation of an XML processor. A Document Type Declaration and any entity declarations that it contains will be escaped when they appear within the content of a `<dsml:value>` element.

4 Matching Capabilities

Having stored some XML-formatted data, it is desirable to be able to search that data for instances matching specified criteria, taking into account the underlying data type for the data. If, for example, some element notionally contains a boolean value, then users would typically like to evaluate whether the element’s value is true or false without regard for the exact syntactic representation. However, if XML-formatted data is stored in an LDAP server using directory attributes with the Directory String or Octet String syntax, then only matching rules that are defined for these syntaxes are available for use in search requests. Since in this context these matching rules are only operating on XML-formatted data at the syntactic level, their usefulness is limited. Consider that the contents of each of the following elements is a valid XML representation of the “true” value for the XML Schema boolean type:

```
<value>>true<\value>
<value>1<\value>
<value> tr&#x75;e <\value>
<value>tr<!-- a comment -->ue<\value>
<value>
  <![CDATA[tr]]>ue
<\value>
```

These examples show only *some* of the potential variations. With such variability at the syntactic

level, the existing character string and octet string matching rules of LDAP are clearly inadequate to do useful matching of XML-formatted data directly. This inadequacy also applies to DSMLv2 and the DSMLv2 Profile of SPMLv2 because they use the same matching rules.

Normalizing XML-formatted data before it is stored in the directory can only remove some of the syntactic variability since Canonical XML and Exclusive XML Canonicalization require that the Infoset [16] representation of the data be preserved. This means that insignificant white space and XML comments have to be retained, among other things. Each of the boolean values in the preceding example has a different Infoset representation and they would have the following form after being transformed by an Infoset-preserving normalization scheme:

```
<value>true<\value>
<value>1<\value>
<value> true <\value>
<value>tr<!-- a comment -->ue<\value>
<value>
  true
<\value>
```

Each of the preceding examples is a single element. In the general case, the data to be matched will be contained by an element or XML attribute nested within a complex XML document. So there is the additional problem of accurately targeting exactly the element or XML attribute to be matched.

A more effective matching strategy would be to extract and normalize the values of certain elements and XML attributes of the XML-formatted data to store in separate directory attributes of a more appropriate directory attribute syntax. Unfortunately, this approach causes a directory server to use more storage space since information in the XML-formatted data is stored redundantly in the separate attributes, and there is always the risk that the values of the separate directory attributes will become inconsistent with the XML-formatted data because of unregulated modifications by unaware directory clients.

The ideal solution is to have matching rules that are aware of the underlying data type of the XML being matched so that insignificant syntactic differences can be ignored. This could be achieved in LDAP on a case-by-case basis by defining new syntaxes and matching rules. However, the component matching rules for LDAP [3] already provide a general capability to match arbitrary parts of structured data. The caveat being that they only apply to data with an underlying ASN.1 type. The XED framework extends the component matching rules so that

they can also be applied to structured data described by an XML Schema or DTD. The component reference that is used in component matching to identify a component to be matched is geared to ASN.1 and isn't flexible enough to identify arbitrary elements or XML attributes in an XML document. XED solves this problem by introducing an alternative method for identifying components that is based on a subset of the XPath abbreviated syntax [21]. The result is that XLDAP is able to perform data-type aware matching of element content in objects.

Like XLDAP, the XSD Profile of SPMLv2, UDDI and the ebXML Registry provide the capability to do data-type aware matching of element content in objects, though they differ in the expressiveness of their query languages.

LDAP search filters, in conjunction with the component matching rules, provide a general-purpose query mechanism that can match instances of arbitrarily chosen parts of objects of arbitrarily chosen object classes. This flexibility is inherited by DSMLv2, the DSMLv2 Profile of SPMLv2 and XLDAP. The XSD Profile of SPMLv2 is similarly general-purpose in its use of XPath expressions to filter objects. XPath expressions in this profile are generally more expressive than component matching, though some of the capabilities of component matching have no equivalent in XPath. Although the XED framework hasn't done so, it would be feasible to define a new matching rule that takes an arbitrary XPath expression as its assertion value and evaluates that expression over the RXER encoding of the directory attribute values to which it is applied.

The filter operations of UDDI and the filter query syntax of the ebXML Registry (one of two query formats that the ebXML Registry offers) have rigidly defined filter structures that are tied to specific predefined object classes. Supporting new object classes or additions to existing object classes actually requires the implementation of protocol extensions, therefore UDDI and the filter query syntax of the ebXML Registry don't have the flexibility or elegance of the query mechanisms of the other protocols. The optionally-supported SQL query syntax of the ebXML Registry offers a way to avoid the limitations of the filter query syntax.

The ability to express queries containing joins between objects varies across the protocols considered in this paper. LDAP (and by inheritance DSMLv2, the DSMLv2 Profile of SPMLv2 and XLDAP) has no support for expressing a join between objects (except in the limited sense that subtree search scope is a recursive join). UDDI only has some predefined

join criteria for relationships between business entity objects. The filter query syntax of the ebXML Registry has a wider range of predefined join criteria for relationships between various classes of object. The XSD Profile of SPMLv2 has a limited, general-purpose join capability through XPath expressions. It is only the SQL query syntax of the ebXML Registry that provides an unrestricted join capability.

XQuery [22] is an extension of XPath that provides an unrestricted join capability. Although the XED framework hasn't done so, it would be feasible to define a new extended operation that takes an arbitrary XQuery expression to evaluate over an XML representation of a subtree of directory entries. XED has not adopted XQuery at this time since it is necessarily more complex and time-consuming to implement than an extension to component matching, and it is not clear that there is a demand for this degree of expressiveness in directory search requests.

5 Object Extensibility

One of the important features of LDAP is the capacity for administrators to extend the directory schema with definitions of new classes of objects (i.e., entry object classes) and definitions of new properties (i.e., directory attributes) for new or existing object classes. This is achieved without having to alter the LDAP server implementation, so LDAP can be said to provide practical support for *user-defined* classes and properties. DSMLv2 and the DSMLv2 Profile of SPMLv2 inherit this ability from LDAP, as does XLDAP, since the XED framework is an extension of the directory model that underlies LDAP. However, whereas LDAP actively discourages the creation of new directory attribute syntaxes (i.e., new property data types), XED embraces the idea and provides the means for administrators to do so. Thus XLDAP can be said to also provide practical support for user-defined property data types.

Objects are modelled by UDDI, the XSD Profile of SPMLv2 and the ebXML Registry as instances of nominated XML Schema types rather than as an abstracted collection of properties. Where an object class is described by an arbitrary XML Schema type, the definition of a new class of object may, in effect, provide the definition of new properties and new property data types. These protocols differ in how readily they can support user-defined object classes.

UDDI search and update protocol operations are tied to specific object classes. For example, there is a separate add operation for each object class. Adding a new class of object requires implementa-

tion in the server of new protocol operations to find and manipulate objects of the class, so UDDI does not provide practical support for user-defined object classes. The extensibility of UDDI is also hampered because it uses Schema Centric Canonicalization for its signed data payloads. The necessity for the signature consumer to know the schemas and embedded languages used by the signature producer means that interworking problems can arise if a UDDI implementation is unilaterally extended.

The search and update protocol operations of the ebXML Registry are not tied to specific object classes, however, the mandatory-to-implement filter query syntax of the Ad Hoc Query operation does have filter expressions that are specific to each object class. Adding a new class of object to the ebXML Registry requires implementation in the server of new filter expression evaluation routines.

The ebXML Registry allows existing object classes to have user-defined properties called slots, but these are limited to just character strings and are matched as such. XML element content is not permitted in a slot, unless it is escaped as in DSMLv2, with the same advantages and disadvantages.

The XSD Profile of SPMLv2 is at least compatible with administrators providing definitions of new classes of objects because the SPMLv2 operations are not tied to specific object classes, though a mechanism to provide new definitions is not specified by SPMLv2.

6 Conclusion

The capabilities of the protocols considered in the previous sections are summarized in Table 1. In each case, support for a feature is considered to be advantageous.

As can be seen from the table, LDAP, DSMLv2 and the DSMLv2 Profile of SPMLv2 are identical in terms of the characteristics considered in this paper.

Unlike LDAP, DSMLv2 and the DSMLv2 Profile of SPMLv2, XLDAP has the single transfer syntax and continuous encoding that are typical of protocols designed around XML as the transfer syntax.

A downside of a continuous encoding is the problem of namespace accumulation, which interferes with Canonical XML. By imposing self-containment on some data payloads, XLDAP obtains the advantage, within a continuous encoding, of compatibility with all the canonical encodings.

A continuous encoding also makes unparsed entities difficult to support (they are trivial to support in LDAP where the encoding is discontinuous). Unparsed entities could be supported in XLDAP with-

Feature	Protocol						
	LDAP	DSMLv2	SPMLv2		UDDI	ebXML Registry	XLDAP
			DSMLv2 Profile	XSD Profile			
Single transfer syntax	no	no	no	yes	yes	yes	yes
Continuous encoding	no	no	no	yes	yes	yes	yes
Supports Canonical XML in payload	yes	yes	yes	no	no	no	yes
Supports Exclusive XML Canonicalization in payload	yes	yes	yes	yes	no	yes	yes
Supports Schema Centric Canonicalization in payload	yes	yes	yes	yes	yes	yes	yes
Supports unparsed entities	yes	yes	yes	no	no	no	no
Data-type aware XML matching	no	no	no	yes	yes	yes	yes
Flexible query expressions	yes	yes	yes	yes	no	maybe	yes
Joins between objects	no	no	no	partial	minimal	yes	no
Allows user-defined object classes	yes	yes	yes	maybe	no	no	yes
Allows user-defined properties (attributes)	yes	yes	yes	maybe	no	yes	yes
Allows user-defined property data types (syntaxes)	no	no	no	maybe	no	no	yes

Table 1: Summary of Protocol Capabilities

out too much difficulty, but being disused, have been prohibited instead, in the interests of simplicity.

The XPath expressions of the XSD Profile of SPMLv2 and the SQL query syntax of the ebXML Registry are more expressive than search requests in XLDAP, though there is no impediment to extending the capabilities of XLDAP in this area.

LDAP has particular strengths in terms of supporting user-defined schema extensions. DSMLv2, the DSMLv2 Profile of SPMLv2 and XLDAP inherit this capability, and the XED framework extends it further to directory attribute syntaxes. In contrast, the schema in UDDI and the ebXML Registry cannot be extended in practice without involving the server vendor. The XSD Profile of SPMLv2 has the potential to support user-defined schema extensions as readily as XLDAP, depending on the implementation.

The general-purpose directory model that underlies LDAP is around twenty years old, but remains relevant today. XLDAP, as a part of the XED framework, realizes that model in a way that compares favourably with much newer, competing, XML-based technologies.

References

- [1] Freed, N. and N. Borenstein, “Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies”, RFC 2045, November 1996.
- [2] Legg, S., “Generic String Encoding Rules (GSER) for ASN.1 Types”, RFC 3641, October 2003.
- [3] Legg, S., “Lightweight Directory Access Protocol (LDAP) and X.500 Component Matching Rules”, RFC 3687, February 2004.
- [4] Zeilenga, K., Ed., “Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map”, RFC 4510, June 2006.
- [5] Legg, S., Ed., “Lightweight Directory Access Protocol (LDAP): Syntaxes and Matching Rules”, RFC 4517, June 2006.
- [6] Legg, S. and D. Prager, “Robust XML Encoding Rules (RXER) for Abstract Syntax Notation One (ASN.1)”, RFC 4910, July 2007.
- [7] Legg, S. and D. Prager, “The XML-Enabled Directory”, draft-legg-xed-roadmap-xx.txt, a work in progress, August 2007.
- [8] Legg, S. and D. Prager, “The XML-Enabled Directory: Protocols”, draft-legg-xed-protocols-xx.txt, a work in progress, August 2007.
- [9] ITU-T Recommendation X.500 (08/05) — ISO/IEC 9594-1:2005, Information technology - Open Systems Interconnection - The Directory: Overview of concepts, models and services.
- [10] ITU-T Recommendation X.509 (08/05) — ISO/IEC 9594-8:2005, Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks.

- [11] ITU-T Recommendation X.680 (07/02) — ISO/IEC 8824-1, Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation.
- [12] ITU-T Recommendation X.690 (07/02) — ISO/IEC 8825-1, Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).
- [13] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielson, H., Thatte, S. and D. Winer, “Simple Object Access Protocol (SOAP) 1.1”, W3C Note, May 2000.
- [14] Boyer, J., “Canonical XML Version 1.0”, W3C Recommendation, March 2001.
- [15] Boyer, J., Eastlake, D. and J. Reagle, “Exclusive XML Canonicalization version 1.0”, W3C Recommendation, July 2002.
- [16] Cowan, J. and R. Tobin, “XML Information Set (Second Edition)”, W3C Recommendation, February 2004.
- [17] Thompson, H., Beech, D., Maloney, M. and N. Mendelsohn, “XML Schema Part 1: Structures Second Edition”, W3C Recommendation, October 2004.
- [18] Biron, P. and A. Malhotra, “XML Schema Part 2: Datatypes Second Edition”, W3C Recommendation, October 2004.
- [19] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E. and F. Yergeau, “Extensible Markup Language (XML) 1.0 (Fourth Edition)”, W3C Recommendation, August 2006.
- [20] Bray, T., Hollander, D., Layman, A. and R. Tobin, “Namespaces in XML 1.0 (Second Edition)”, W3C Recommendation, August 2006.
- [21] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J. and J. Simeon, “XML Path Language (XPath) 2.0”, W3C Recommendation, January 2007.
- [22] Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J. and J. Simeon, “XQuery 1.0: An XML Query Language”, W3C Recommendation, January 2007.
- [23] “Directory Services Markup Language v2.0”, OASIS Standard, December 2001.
- [24] Clement, L., Hatley, A., von Riegen, C. and T. Rogers, “UDDI Version 3.0.2”, OASIS Technical Committee Draft, October 2004.
- [25] Moses, T., Ed., “eXtensible Access Control Markup Language (XACML) Version 2.0”, OASIS Standard, February 2005.
- [26] Fuger, S., Najmi, F. and N. Stojanovic, “ebXML Registry Services and Protocols Version 3.0”, OASIS Standard, May 2005.
- [27] Atkinson, B., Aissi, S., Hatley, A. and M. Hondo, “Schema Centric XML Canonicalization Version 1.0”, OASIS Technical Committee Specification, May 2005.
- [28] Cole, G., Ed., “OASIS Service Provisioning Markup Language (SPML) Version 2”, OASIS Standard, April, 2006.
- [29] Bohren, J., Ed., “OASIS Service Provisioning Markup Language (SPML) v2 - DSML v2 Profile”, OASIS Standard, April, 2006.
- [30] Bohren, J., Ed., “OASIS Service Provisioning Markup Language (SPML) v2 - XSD Profile”, OASIS Standard, April, 2006.